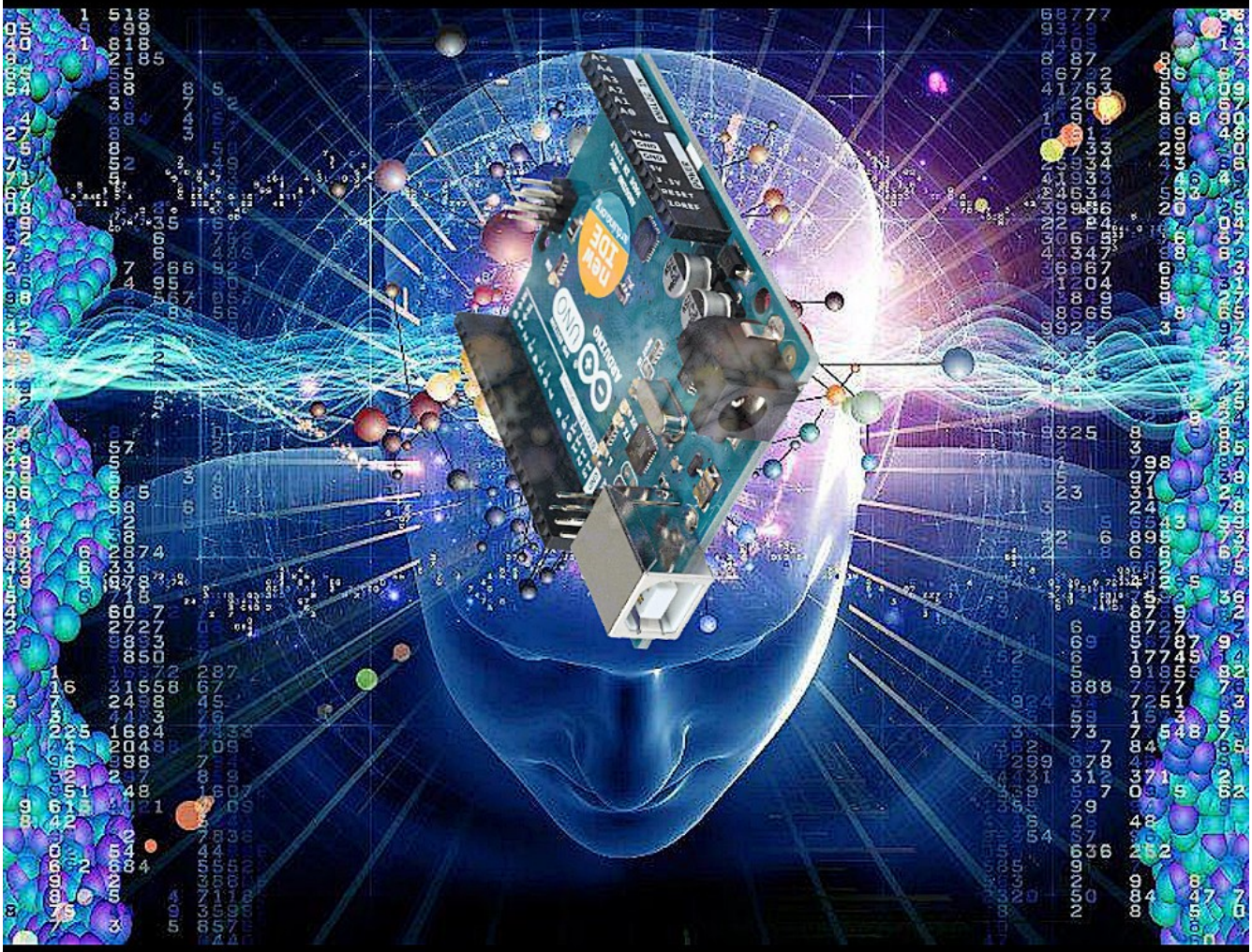
Ultimate Arduino Handbook



Neurohacker's Edition, v1.0

"You either hack your own mind and brain or they get hacked for you."

- Dr. Zachary Stein

Table Of Contents

Part 2: Digital Electronics Essentials

Part 3: Handy Tables, Charts & Formulas

Misc. Conversions & Formulas

Legal Notices & Warnings

Legal Notices:

You are permitted to remix this report provided that:

- You give *prominent credit and link love* where credit's due, and prominently pass these same CC BY SA rights onto others within your remix;

- You use a *completely different title and ecover design*, and **do not** in any way insinuate that Mark Maffei endorses your remix in any manner whatsoever.

The Publisher of this report has striven to be as accurate and complete as possible in its creation. However, *this does not warrant or represent at any time that the contents within are suitable for all purposes and intents.*

As a reader of this report, you understand and agree to the following; if not, *simply discontinue using this report*:

- The Publisher and individual authors of this report *will not be responsible for any losses or damages of any kind incurred by the reader*; whether directly or indirectly arising from the use of the information found in this report. Anything you do with the following information is **at your own personal risk**.

- The Publisher of this report reserves the right to make changes *without notice at his sole discretion and assumes no responsibility or liability whatsoever* on the behalf of the reader of this report.

- Readers are advised to *exercise due diligence and seek professional help when prudent to do so.*

Warnings:

- Children are advised *against* brainwave entrainment, as the brain is still developing and is more sensitive.

- If you are prone to seizures/epilepsy or have an irregular heartbeat (especially if you have a pacemaker), you are advised *against* brainwave entrainment.

- Pregnant women should *consult their physician* when considering using brainwave entrainment.

Simply Stated: *This information is for educational, entertainment, and informational research purposes only.*

Acknowledgements & Attribution

First, a salute to the following brilliant brainhackers and Master Circuit Warriors:

- Lady Ada
- Eric Mika
- Alex Glow
- R.O. Jones
- Bill Wilson
- Nate Seidle
- Sophi Kravitz
- Mitch Altman
- Don Lancaster
- Bobby Cossum
- Tony R. Kuphaldt
- Robert Grossblatt
- Forrest M. Mims III
- Darren Mothersele

A hearty "Thank you!" especially goes out to Adafruit, Arduino.cc, Makezine.com, Hackster.io and SparkFun in the *ongoing research and development* of this work.

Attribution: The *Ultimate Arduino Handbook, Neurohacker's Edition* is the passionate union of the original Ultimate Arduino Handbook+ the original Ultimate Arduino Brainhacker's Handbook, both compiled by Mark Maffei and warmly dedicated to Arduino aficionados of *all persuasions*: May your Arduino adventures always be pleasurably rewarding!

Redistribute generously!

Brief Word From The Editor

"*The brain is one of the final frontiers of human discovery. Each day it gets easier to leverage technology to expand the capabilities of that squishy thing inside our heads. Real-world BCI will be vital in reverse-engineering and further understanding the human brain.*"

- Conor Russomanno, OpenBCI co-founder

From my very first Radio Shack 150-in-1 Electronics Experimenter Kit, back when I was 9 or 10, I was instantly hooked on what is now termed the Makerspace/Hackerspace. From there, it was onto motorized Legos, robotic arms, and ultimately the “computer revolution” of the 80’s (which admittedly dates me).

However, it wasn’t until the 90’s that I first started seriously experimenting with what later became known as “neurohacking”, “brainhacking”, "consciousness hacking", etc.

I hand-crafted a set of homemade external brain-signal sensors (what I simply referred to as “trodes”) made out of a modified cheap pair of headphones, a couple of silver dimes and a low-noise op-amp circuit configured as a low-pass brainwave amplifier.

I still fondly recall the first time I saw my own brainwaves on my trusty old ‘scope, and then watching them change as I changed my concentration/focus.

Now fast forward to 2011 - my initial discovery of the **Arduino UNO**: Love at first sight!

Economical, yet powerful microcontrollers like Arduino and readily available EEG toys like NeuroSky's MindFlex make brain-computer interface **(BCI)** neurohacking *very analogous* to taking the **red pill** in the Matrix, in a certain manner of speaking.

Except in this case... focused, clear, productive and fully present, with higher cognition, faster information processing and easier concentration... these are *just some of* the amazing results brainhackers from around the world have experienced.

However, finding high quality information was the real issue. Sure, there’s undoubtedly lots of great information on Arduino, Arduino-based neurohacking and Electronics… spread out all over the place and *never conveniently accessible* right when you actually need it.

In stepping up to the plate and attaining to provide a handy solution to this problem, the Ultimate Arduino Handbook is an ever-evolving labor of love that aims to be a monthly publication.

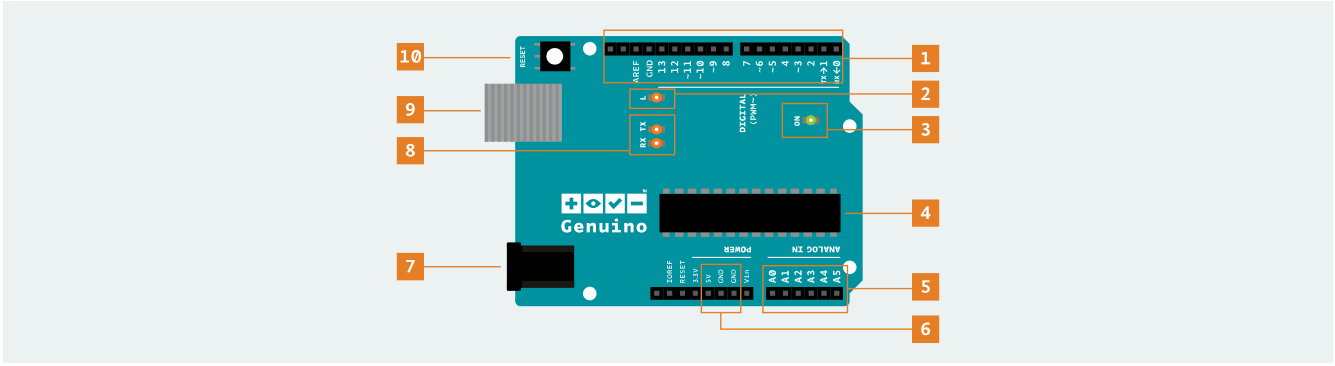
Nonetheless, enjoy!

Best regards,

Mark Maffei

Arduino Essentials

Arduino/Genuino UNO Board Anatomy



This image was developed using Fritzing. For more circuit examples, see the Fritzing project page

1. **Digital pins**: Use these pins with digitalRead(), digitalWrite(), and analogWrite(). analogWrite() works only on the pins with the PWM symbol.

2. **Pin 13 LED**: The only actuator built-in to your board. Besides being a handy target for your first blink sketch, this LED is very useful for debugging.

3. **Power LED**: Indicates that your Genuino is receiving power. Useful for debugging.

4. **Atmega microcontroller**: The heart of your board.

5. **Analog in**: Use these pins with analogRead().

6. **GND and 5V pins:** Use these pins to provide +5V power and ground to your circuits.

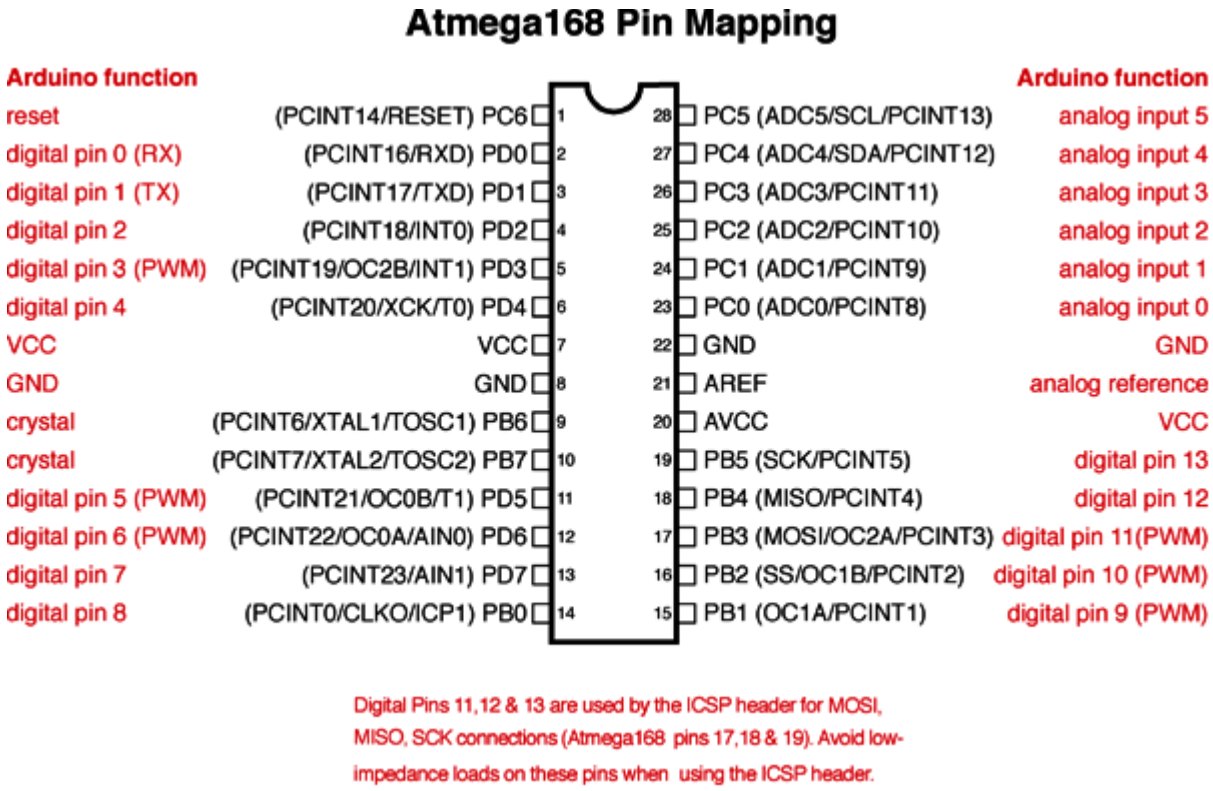
7. **Power connector**: This is how you power your Genuino when it's not plugged into a USB port for power. Can accept voltages between 7-12V.

8. **TX and RX LEDs**: These LEDs indicate communication between your Genuino and your computer. Expect them to flicker rapidly during sketch upload as well as during serial communication. Useful for debugging.

9. **USB port**: Used for powering your Genuino Uno, uploading your sketches to your Genuino, and for communicating with your Genuino sketch (via Serial. println() etc.).

10. **Reset button**: Resets the Atmega microcontroller.

Arduino Brain Pinout:



Digital Pins

The pins on the Arduino can be configured as either inputs or outputs. This document explains the functioning of the pins in those modes. While the title refers to digital pins, it is important to note that vast majority of Arduino (Atmega) analog pins, may be configured, and used, in exactly the same manner as digital pins.

Properties of Pins Configured as INPUT

Arduino (Atmega) pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode() when you're using them as inputs. Pins configured this way are said to be in a **high-impedance state**.

Input pins make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100MΩ in front of the pin.

This means that it takes very little current to move the input pin from one state to another, and can make the pins useful for such tasks as implementing a capacitive touch sensor, reading an LED as a photodiode, or reading an analog sensor with a scheme such as RCTime.

This also means however, that pins configured as `pinMode(pin, INPUT)` with nothing connected to them (or with wires not connected to another circuit) will report seemingly random changes in pin state (via electrical noise from the environment, or capacitively coupling the state of a nearby pin).

Pullup Resistors With Pins Configured As INPUT

Often it is useful to steer an input pin to a known state if no input is present. This can be done by adding a pullup resistor (to +5V), or a pulldown resistor (resistor to ground) on the input.

A **10KΩ resistor** is a good value for a pullup or pulldown resistor.

Properties Of Pins Configured As INPUT_PULLUP

There are 20KΩ pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed by setting the pinMode() as `INPUT_PULLUP`.

This effectively inverts the behavior of the `INPUT` mode, where `HIGH` means the sensor is off, and `LOW` means the sensor is on.

The value of this pullup depends on the microcontroller used. On most AVR-based boards, the value is guaranteed to be between 20kΩ and 50kΩ. On the Arduino Due, it is between 50kΩ and 150kΩ. For the exact value, consult the datasheet of the microcontroller on your board.

When connecting a sensor to a pin configured with `INPUT_PULLUP`, the other end should be connected to ground. In the case of a simple switch, this causes the pin to read `HIGH` when the switch is open, and `LOW` when the switch is pressed.

The pullup resistors provide enough current to dimly light an LED connected to a pin that has been configured as an input. If LEDs in a project seem to be working, but very dimly, this is likely what is going on.

The pullup resistors are controlled by the same registers (internal chip memory locations) that control whether a pin is `HIGH` or `LOW`.

Consequently, a pin that is configured to have pullup resistors turned on when the pin is an `INPUT` will have the pin configured as `HIGH` if the pin is then switched to an OUTPUT with pinMode().

Conversely, an output pin that is left in a `HIGH` state will have the pullup resistors set if switched to an input with pinMode().

Note: Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards.

If you enable its internal 20k pull-up resistor, it will hang at around 1.7V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down. In other words, it always returns `LOW`. If you must use pin 13 as a digital input, set its pinMode() to `INPUT` and use an external pull down resistor.

Properties Of Pins Configured As OUTPUT

Pins configured as `OUTPUT` with pinMode() are said to be in a **low-impedance state**. This means that they can provide a substantial amount of current to other circuits.

Atmega pins can **source** (provide positive current) or **sink** (provide negative current) up to 20mA (relatively safely) and 40 mA (MAX) of current to other devices/circuits.

For example, this is enough current to brightly light up an LED (with a series resistor), or run several sensors; albeit not enough current to run most relays, solenoids, or motors.

Short circuits on Arduino pins (or attempting to run high current devices from them) can damage or destroy the output transistors in the pin, or possibly the entire Arduino.

Short circuits often result in a "dead" pin in the microcontroller; yet the remaining chip will still function adequately. For this reason it is a good idea to connect `OUTPUT` pins to other devices with **470Ω or 1kΩ resistors**, unless maximum current draw from the pins is absolute required for a particular application. If so – *proceed with caution.*

Further Learning: pinMode(), digitalWrite(), digitalRead()

Analog Input Pins

What follows is a description of the analog input pins on an Arduino chip (Atmega8, Atmega168, Atmega328, or Atmega1280).

A/D Converter

The Atmega controllers used for the Arduino contain an onboard **6 channel analog-to-digital (A/D) converter** with 10 bit resolution, returning integers from 0 to 1023.

While the main function of the analog pins for most Arduino users is to read analog sensors, the analog pins also have all the functionality of general purpose input/output (**GPIO**) pins (the same as digital pins 0 - 12).

Consequently, if a user needs more general purpose input output pins, and all the analog pins are not in use, the analog pins may be used for GPIO.

Pin Mapping

The analog pins can be used identically to the digital pins, using the aliases A0 (for analog input 0), A1, etc.

For example, the code would look like this to set analog pin 0 to an output, and to set it `HIGH`:

```
pinMode(A0, OUTPUT);

digitalWrite(A0, HIGH);
```

Pullup Resistors

The analog pins also have pullup resistors, which work identically to pullup resistors on the digital pins. They are enabled by issuing a command such as:

```
digitalWrite(A0, INPUT_PULLUP);  // set pullup on analog pin 0
```

Tips & Caveats

- Turning on a pullup will affect the values reported by `analogRead()`.
- The `analogRead` command will not work correctly if a pin has been previously set to an output; so if this is the case, set it *back to an input* before using `analogRead`.
- Similarly if the pin has been set to `HIGH` as an output, the pullup resistor will be set, when switched back to an input.
- The Atmega datasheet also *cautions against switching analog pins in close temporal proximity* to making A/D readings (`analogRead`) on other analog pins. This can cause electrical noise and introduce jitter in the analog system.
- It may be desirable, after manipulating analog pins (in digital mode), to add a *short delay* before using analogRead() to read other analog pins.

Memory

The notes on this page are for *all boards* **except the Due,** which has a different architecture.

There are three pools of memory in the microcontroller used on avr-based Arduino boards:

- **Flash memory** is program space where the Arduino sketch is stored.

- **SRAM** (static random access memory) is where the sketch creates and manipulates variables when it runs.

- **EEPROM** (electrically erasable programmable read-only memory) is memory space that programmers can use to store long-term information.

Flash memory and EEPROM memory are *non-volatile* (i.e. the information persists after the power is turned off). However, SRAM is volatile and will be lost when the power is cycled.

- The ATmega328 chip found on the UNO has the following amounts of memory: Flash - 32kb (of which 0.5bk is used for the bootloader), SRAM - 2kb, and EEPROM – 1kb.

- The ATmega2560 in the Mega2560 has larger memory space: Flash - 256k bytes (of which 8k is used for the bootloader), SRAM - 8kb, and EEPROM – 4kb.

Notice that there's not much SRAM available in the UNO. It's easy to use it all up by having lots of strings in your program. For example, a declaration like: `char message[] = "I support the Cape Wind project.";` puts 33 bytes into SRAM (each character takes a byte, plus the '\0' terminator).

This might not seem like a lot, but it doesn't take long to get to 2048, especially if you have a large amount of text to send to a display, or a large lookup table, for example. If you run out of SRAM, your program may fail in unexpected ways; it will appear to upload successfully, but not run, or run strangely.

To check if this is happening, you can try commenting out or shortening the strings or other data structures in your sketch (without changing the code). If it then runs successfully, you're probably running out of SRAM.

There are a few things you can do to address this problem:

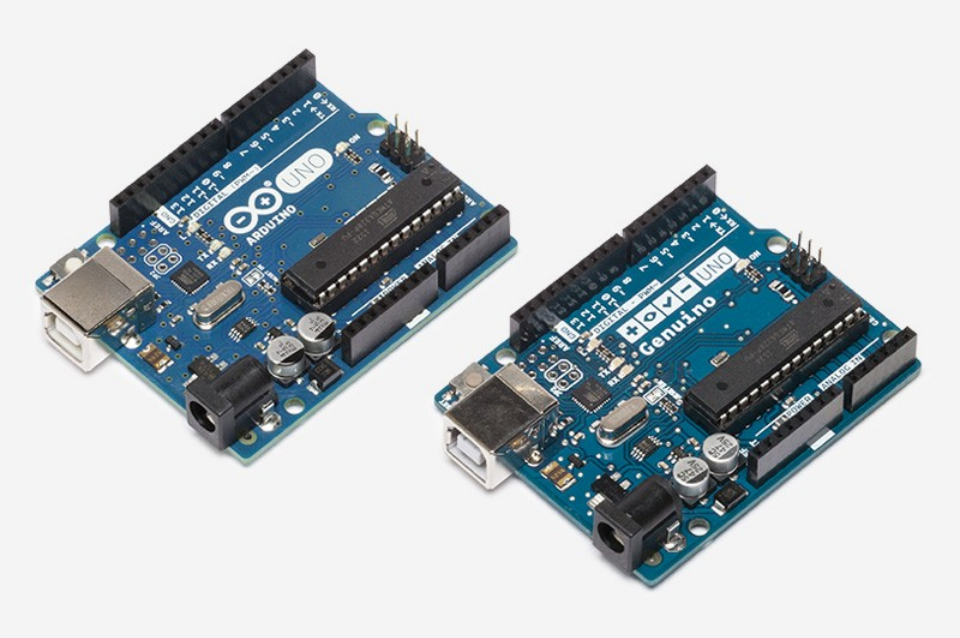
- If your sketch talks to a program running on an external computer, you can try shifting data or calculations to the computer, reducing the load on the Arduino.

- If you have lookup tables or other large arrays, use the smallest data type necessary to store the values you need; for example, an int takes up two bytes, while a byte uses only one (but can store a smaller range of values).

- If you don't need to modify the strings or data while your sketch is running, you can store them in flash (program) memory instead of SRAM; to do this, use the PROGMEM keyword.

To use the EEPROM, see the EEPROM library.

Arduino IDE Basics



Attribution: Arduino.cc

The Arduino Integrated Developer Environment (IDE) software allows you to write programs and upload them to your board.

On the Arduino Software page you will find two options:

1. If you have a reliable Internet connection, you should use the online IDE (Arduino Web Editor).

It will allow you to save your sketches in the cloud, having them available from any device and backed up. You will always have the most up-to-date version of the IDE without the need to install updates or community generated libraries.

To use the online IDE simply follow these instructions. Remember that boards work out-of-the-box on the Web Editor, no need to install anything.

2. If you would rather work offline, you should use the latest version of the desktop IDE.

To get step-by-step instructions select one of the following link accordingly to your operating system:

- Windows
- Mac OS X
- Linux

- Portable IDE (Windows and Linux)

The USB cable shown below is necessary to program the board and not just to power it up:

Attribution: Arduino.cc

The UNO automatically draw power from either the USB or an external power supply. Connect the board to your computer using the USB cable. The green power LED (labelled **PWR**) should go on.

Windows Installer Considerations

If you used the Windows Installer (XP up to 10), it will install drivers automatically as soon as you connect your board.

If you downloaded and expanded the Zip package or, for some reason, *the board wasn't properly recognized*, please follow the procedure below:

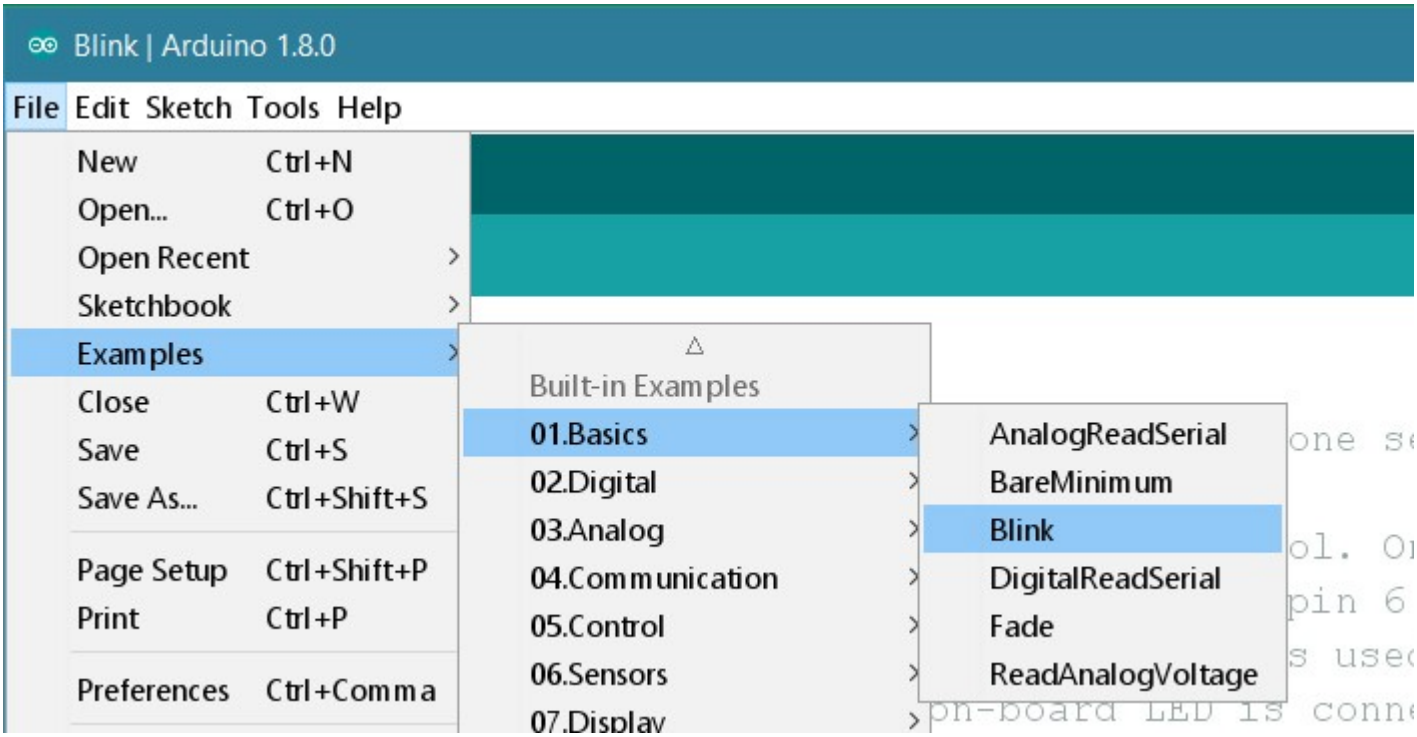
- Click on the Start Menu, and open up the Control Panel.
- While in the Control Panel, navigate to System and Security. Next, click on System. Once the System window is up, open the Device Manager.
- Look under Ports (COM & LPT). You should see an open port named "Arduino UNO (COMxx)". If there is no COM & LPT section, look under "Other Devices" for "Unknown Device".

- Right click on the "Arduino UNO (COmxx)" port and choose the "Update Driver Software" option.
- Next, choose the "Browse my computer for Driver software" option.
- Finally, navigate to and select the driver file named **"arduino.inf"**, located in the "Drivers" folder of the Arduino Software download (not the "FTDI USB Drivers" sub-directory). If you are using an old version of the IDE (1.0.3 or older), choose the UNO driver file named **"Arduino UNO.inf"**

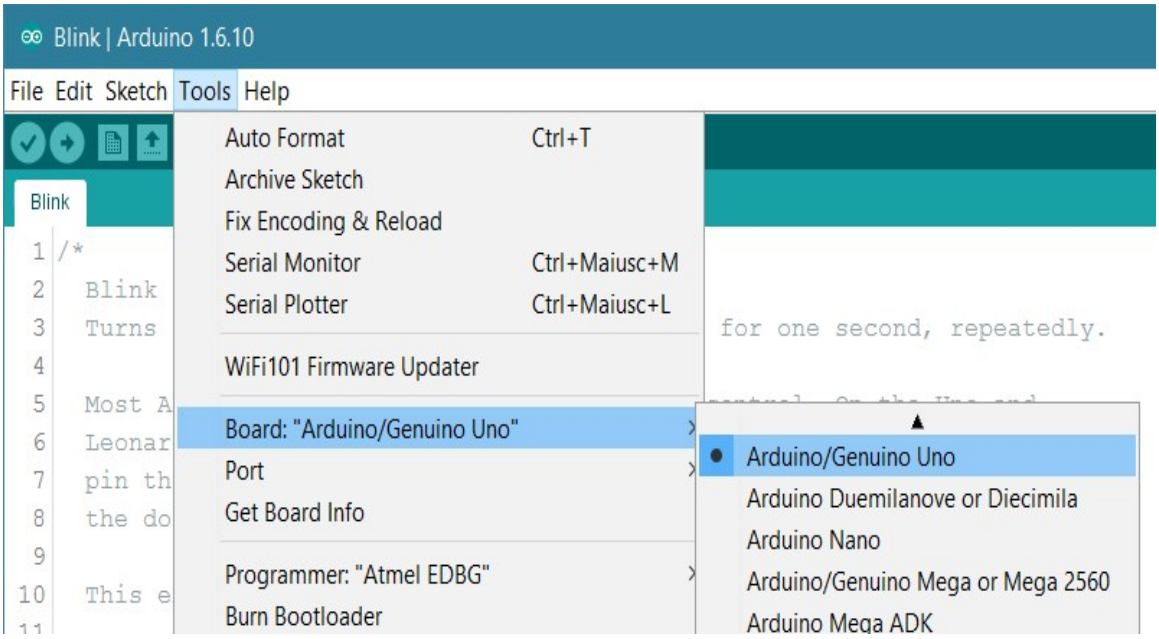
Windows will finish up the driver installation from there. See also: step-by-step screenshots for installing the Uno under Windows XP.

Open Your First Sketch

Open the LED blink example sketch: **File > Examples >01.Basics > Blink**.

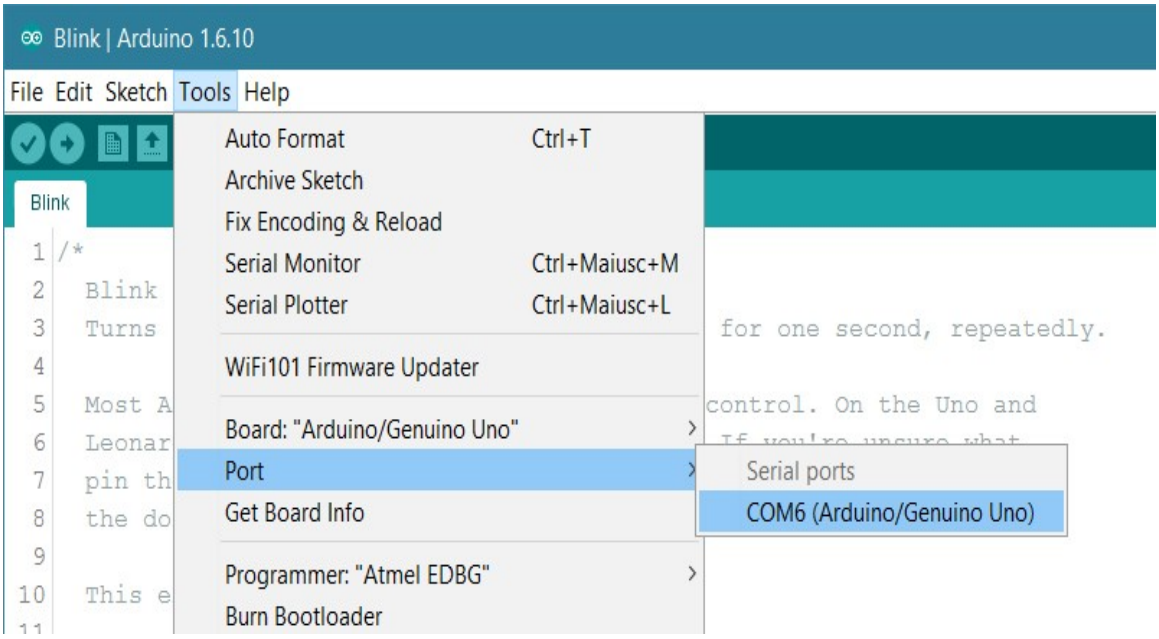


Next, select the entry in the **Tools > Board** menu that matches your Arduino/Genuino board.



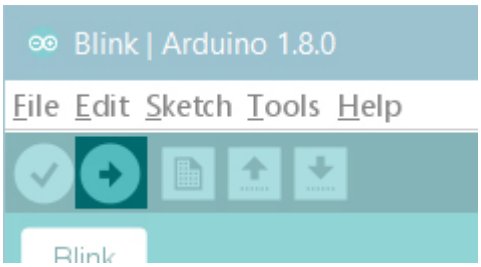
Select the serial device of the board from the Tools | Serial Port menu. This is likely to be **COM3** or higher (**COM1** and **COM2** are usually reserved for hardware serial ports).

To find out, you can disconnect your board and re-open the menu; the entry that disappears should be the Arduino or Genuino board. Reconnect the board and select that serial port.



Now, simply click the "Upload" button in the environment. Wait a few seconds - you should see the RX and TX leds on the board flashing.

If the upload is successful, the message "Done uploading." will appear in the status bar.

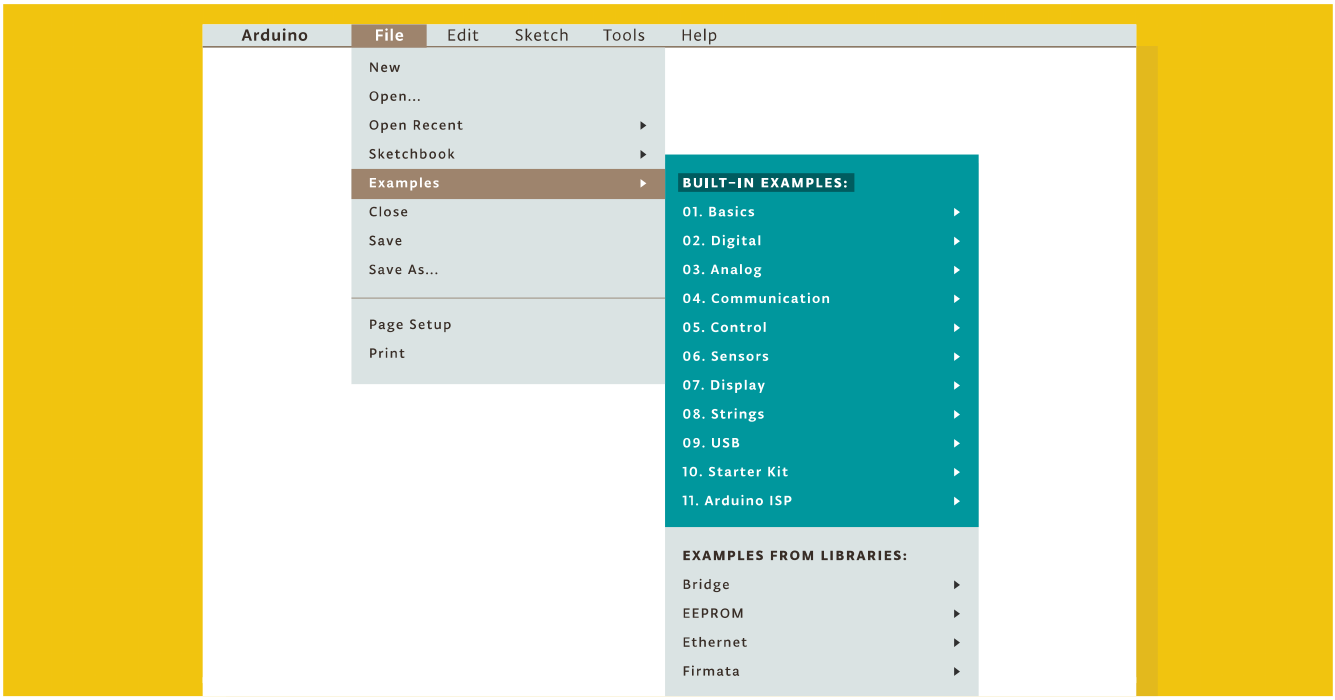


A few seconds after the upload finishes, you should see the pin 13 (L) LED on the board start to blink (in orange). If it does, congratulations! You've gotten Arduino or Genuino up and running.

If you have problems, please see the troubleshooting suggestions.

You can learn more about the Desktop IDE via this generic guide on the Arduino IDE with a few more infos on the Preferences, the Board Manager, and the Library Manager.

Built-in Examples are sketches included in the Arduino Software (IDE), to open them click on the toolbar menu: **File > Examples**. These simple programs demonstrate all basic Arduino commands. They span from a Sketch Bare Minimum to Digital and Analog IO, to the use of Sensors and Displays.



Arduino IDE Anatomy

The Arduino Integrated Development Environment (IDE) contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus. It connects to the Arduino and Genuino hardware to upload programs and communicate with them.

Writing Sketches

Programs written using Arduino IDE are called **sketches**. These sketches are written in the text editor and are saved with the file extension **.ino**. The editor has features for cutting/ pasting and for searching/replacing text.

The message area gives feedback while saving and exporting and also displays errors. The console displays text output by the Arduino IDE, including complete error messages and other information.

The bottom right-hand corner of the window displays the configured board and serial port. The toolbar buttons allow you to verify and upload programs, create, open, and save sketches, and open the serial monitor.

NB: Versions of the Arduino IDE prior to 1.0 saved sketches with the extension .pde. It is still possible to open v1.0 files, albeit you will be prompted to save the sketch with the .ino extension on save.

Verify
Checks your code for errors compiling it.

Upload
Compiles your code and uploads it to the configured board. See uploading below for details. **Note:** If you are using an external programmer with your board, you can hold down the "shift" key on your computer when using this icon. The text will change to "Upload using Programmer"

New
Creates a new sketch.

Open
Presents a menu of all the sketches in your sketchbook. Clicking one will open it within the current window overwriting its content. **Note:** due to a bug in Java, this menu doesn't scroll; if you need to open a sketch late in the list, use the **File | Sketchbook** menu instead.

Save

Saves your sketch.

Serial Monitor

Opens the serial monitor.

Additional commands are found within the five menus: **File**, **Edit**, **Sketch**, **Tools**, and **Help**.

The menus are *context sensitive*, which means only those items relevant to the work currently being carried out are available.

File

- **New**: Creates a new instance of the editor, with the bare minimum structure of a sketch already in place.
- **Open**: Allows to load a sketch file browsing through the computer drives and folders.
- **Open Recent**: Provides a short list of the most recent sketches, ready to be opened.
- **Sketchbook**: Shows the current sketches within the sketchbook folder structure; clicking on any name opens the corresponding sketch in a new editor instance.
- **Examples**: Any example provided by the Arduino Software (IDE) or library shows up in this menu item. All the examples are structured in a tree that allows easy access by topic or library.
- **Close**: Closes the instance of the Arduino Software from which it is clicked.
- **Save**: Saves the sketch with the current name. If the file hasn't been named before, a name will be provided in a "Save as.." window.
- **Save as...** Allows to save the current sketch with a different name.
- **Page Setup**: It shows the Page Setup window for printing.
- **Print**: Sends the current sketch to the printer according to the settings defined in Page Setup.
- **Preferences**: Opens the Preferences window where some settings of the IDE may be customized, as the language of the IDE interface.
- **Quit**: Closes all IDE windows. The same sketches open when Quit was chosen will be automatically reopened the next time you start the IDE.

Edit

- **Undo/Redo**: Goes back of one or more steps you did while editing; when you go back, you may go forward with Redo.
- **Cut**: Removes the selected text from the editor and places it into the clipboard.
- **Copy**: Duplicates the selected text in the editor and places it into the clipboard.
- **Copy for Forum**: Copies the code of your sketch to the clipboard in a form suitable for posting to the forum, complete with syntax coloring.
- **Copy as HTML**: Copies the code of your sketch to the clipboard as HTML, suitable for embedding in web pages.
- **Paste**: Puts the contents of the clipboard at the cursor position, in the editor.
- **Select All**: Selects and highlights the whole content of the editor.
- **Comment/Uncomment**: Puts or removes the // comment marker at the beginning of each selected line.
- **Increase/Decrease Indent**: Adds or subtracts a space at the beginning of each selected line, moving the text one space on the right or eliminating a space at the beginning.
- **Find**: Opens the Find and Replace window where you can specify text to search inside the current sketch according to several options.
- **Find Next**: Highlights the next occurrence - if any - of the string specified as the search item in the Find window, relative to the cursor position.
- **Find Previous**: Highlights the previous occurrence - if any - of the string specified as the search item in the Find window relative to the cursor position.

Sketch

- **Verify/Compile**: Checks your sketch for errors compiling it; it will report memory usage for code and variables in the console area.
- **Upload**: Compiles and loads the binary file onto the configured board through the configured Port.
- **Upload Using Programmer**: This will overwrite the bootloader on the board; you will need to use *Tools > Burn Bootloader* to restore it and be able to Upload to USB serial

port again. However, it allows you to use the full capacity of the Flash memory for your sketch. **Note:** This command will NOT burn the fuses. To do so a *Tools -> Burn Bootloader* command must be executed.

- **Export Compiled Binary**: Saves a .hex file that may be kept as archive or sent to the board using other tools.
- **Show Sketch Folder**: Opens the current sketch folder.
- **Include Library***:* Adds a library to your sketch by inserting #include statements at the start of your code. For more details, see *Libraries* below. Additionally, from this menu item you can access the Library Manager and import new libraries from .zip files.
- **Add File…** Adds a source file to the sketch (it will be copied from its current location). The new file appears in a new tab in the sketch window. Files can be removed from the sketch using the tab menu, accessible by clicking on the small triangle icon below the serial monitor one on the right side o the toolbar.

Tools

- **Auto Format**: This formats your code nicely (i.e. indents it so that opening and closing).
- **Archive Sketch**: Archives a copy of the current sketch in .zip format. The archive is placed in the same directory as the sketch.
- **Fix Encoding & Reload**: Fixes possible discrepancies between the editor char map encoding and other operating systems char maps.
- **Serial Monitor**: Opens the serial monitor window and initiates the exchange of data with any connected board on the currently selected Port. This usually resets the board, if the board supports Reset over serial port opening.
- **Board**: Select the board that you're using. See below for descriptions of the various boards.
- **Port**: This menu contains all the serial devices (real or virtual) on your machine. It should automatically refresh every time you open the top-level tools menu.
- **Programmer**: For selecting a hardware programmer when programming a board or chip and not using the onboard USB-serial connection. Normally you won't need this, but if you're burning a bootloader to a new microcontroller, you will use this.
- **Burn Bootloader**: The items in this menu allow you to burn a bootloader onto the microcontroller on an Arduino board. This is not required for normal use of an

Arduino or Genuino board; but is useful if you purchase a new Atmega micro-controller that comes without a bootloader.

Note: Ensure that you've selected the correct board from the **Boards** menu before burning the bootloader on the target board. This command also set the right fuses.

Help

Here you find easy access to a number of documents that come with the Arduino IDE. You have access to Getting Started, Reference, this guide to the IDE and other documents locally, without an Internet connection. The documents are a local copy of the online ones and may link back to our online website.

Find in Reference: This is the only interactive function of the Help menu; it directly selects the relevant page in the local copy of the Reference for the function or command under the cursor.

Sketchbook

The Arduino IDE uses the concept of a sketchbook: a standard place to store your programs (aka "**sketches**"). The sketches in your sketchbook can be opened from the *File > Sketchbook* menu or from the *Open button* on the toolbar.

The first time you run the Arduino software, it will automatically create a directory for your sketchbook. You can view or change the location of the sketchbook location from with the *Preferences* dialog.

Beginning with version 1.0, files are no longer saved as a .pde file, but can still be opened; automatically renamed with an .ino when saved.

Tabs, Multiple Files, and Compilation

Allows you to manage sketches with more than one file (each of which appears in its own tab). These can be normal Arduino code files (no visible extension), C files (.c extension), C++ files (.cpp), or header files (.h).

Uploading

Before uploading your sketch, you need to select the correct items from the *Tools > Board* and *Tools > Port* menus. The boards are described below.

- On the Mac, the serial port is probably something like **/dev/tty.usbmodem241** (for an UNO or Mega2560 or Leonardo) or **/dev/tty.usbserial-1B1** (for a Duemilanove or earlier USB board), or **/dev/tty.USA19QW1b1P1.1** (for a serial board connected with a Keyspan USB-to-Serial adapter).

- On Windows, it's probably **COM1** or **COM2** (for a serial board) or **COM4**, **COM5**, **COM7**, or higher (for a USB board). To find out, you look for USB serial device in the ports section of the Windows Device Manager.

- On Linux, it should be **/dev/ttyACMx**, **/dev/ttyUSBx** or similar.

Once you've selected the correct serial port and board, press the upload button in the toolbar or select Upload from the *Sketch menu*. Current Arduino boards will reset automatically and begin the upload.

With older pre-Diecimila boards that lack auto-reset, you'll need to press the reset button on the board just before starting the upload.

On most boards, you'll see the RX and TX LEDs blink as the sketch is uploaded. The Arduino IDE will display a message when the upload is complete, or show an error.

When you upload a sketch, you're using the Arduino **bootloader**; a small program that has been loaded on to the microcontroller on your board. It allows you to upload code without using any additional hardware.

The bootloader is active for a few seconds when the board resets; then it starts whichever sketch was most recently uploaded to the microcontroller. The bootloader will blink the on-board (pin 13) LED when it starts (i.e. when the board resets).

Libraries

Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data.

To use a library in a sketch, select it from the *Sketch > Import Library* menu. This will insert one or more `#include` statements at the top of the sketch and compile the library with your sketch.

Because libraries are uploaded to the board with your sketch, they increase the amount of space it takes up. If a sketch no longer needs a library, simply delete its `#include` statements from the top of your code.

There is a list of libraries in the reference.

Some libraries are included with the Arduino software. Others can be downloaded from a variety of sources or through the Library Manager. Starting with version 1.0.5 of the IDE, you do can import a library from a zip file and use it in an open sketch. See these instructions for installing a third-party library. To write your own library, see this tutorial.

Third-Party Hardware

Support for third-party hardware can be added to the *hardware directory* of your sketchbook directory. Platforms installed there may include board definitions (which appear in the board menu), core libraries, bootloaders, and programmer definitions.

- To install, create the *hardware directory*, then unzip the third-party platform into its own sub-directory. **Note:** Do not use "arduino" as the sub-directory name or you'll override the built-in Arduino platform.)

- To uninstall, simply delete its directory. For details on creating packages for third-party hardware, see the Arduino IDE 1.5 3rd party Hardware specification.

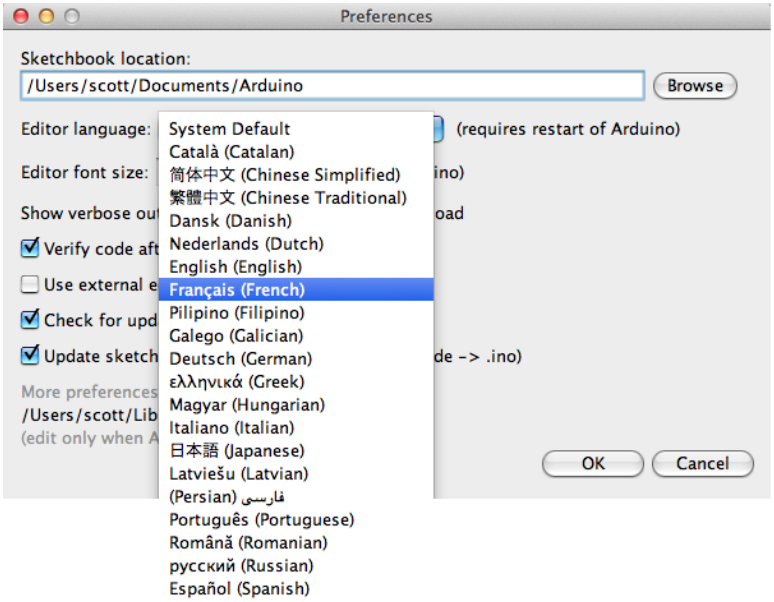
Serial Monitor

Displays serial data being sent from the Arduino or Genuino board (USB or serial board). To send data to the board, enter text and click on the "send" button or press enter.

Choose the baud rate from the drop-down that matches the rate passed to **Serial.begin** in your sketch. **Note:** On Windows, Mac or Linux, the Arduino or Genuino board will reset (rerun your sketch execution to the beginning) when you connect with the serial monitor.

You can also talk to the board from Processing, Flash, MaxMSP, etc (see the interfacing page for details).

Language Support



Since version 1.0.1 , the Arduino IDE has been translated into 30+ different languages. By default, the IDE loads in the language selected by your operating system.

Note: on Windows and possibly Linux, this is determined by the locale setting which controls currency and date formats, not by the language the OS is displayed in.

If you would like to change the language manually:

1. Start the Arduino IDE and open the **Preferences** window.

2. Next to the **Editor Language** there is a dropdown menu of currently supported languages. Select your preferred language from the menu, and restart the software to use the selected language.

If your OS language is not supported, the Arduino IDE will default to English. You can return the software to its default setting of selecting its language based on your operating system by selecting **System Default** from the **Editor Language** drop-down.

This setting will take effect when you restart the Arduino IDE. Similarly, after changing your operating system's settings, you must restart the Arduino IDE to update it to the new default language.

Preferences

Some preferences can be set in the preferences dialog (found under the **Arduino** menu on the Mac, or **File** on Windows and Linux). The rest can be found in the preferences file, whose location is shown in the preference dialog.

Boards

The board selection has two effects: it sets the parameters (e.g. CPU speed and baud rate) used when compiling and uploading sketches; and sets and the file and fuse settings used by the burn bootloader command.

Some of the board definitions differ only in the latter, so even if you've been uploading successfully with a particular selection you'll want to check it before burning the bootloader. You can find a comparison table between the various boards here.

Arduino IDE includes the built in support for the boards in the following list, all based on the AVR Core. The Boards Manager included in the standard installation allows to add support for the growing number of new boards based on different cores like Arduino Due, Arduino Zero, Edison, Galileo and so on.

Sketch Basics

In this tutorial, you'll learn a few core sketch basics.

Comments

Comments are ignored by the sketch. They're included for people reading the code: to explain what the program does, how it works and/or why it's written the way it is.

It's a good practice to comment your sketches, and to keep the comments up-to-date when you modify the code; albeit not sloppily, as comments do take up memory resources.

The first type of *comment*:

```
/* Yadayadayada...
 *
 * Blahblahblah...
 */
```

Everything between the `/*` and `*/` is ignored by the Arduino when it runs the sketch. The * at the start of each line is only there for clarity, not required. There's also another style for short **single-line comments**. These start with `//` and continue to the end of the line. For example, in the line:

```
int ledPin = 13;                // LED connected to digital pin 13
```

The message "LED connected to digital pin 13" is a comment.

Variables

A *variable* is a place for storing a piece of data. It has a name, a type, and a value. For example, the line from the Blink sketch above declares a variable with the name `ledPin`, the type `int`, and an initial value of `13`.

It's being used to indicate which Arduino pin the LED is connected to. Every time the name `ledPin` appears in the code, its value will be retrieved. In this case, the person writing the program could have chosen not to bother creating the `ledPin` variable and instead have simply written 13 everywhere they needed to specify a pin number.

However, the advantage of using a variable is that it's easier to move the LED to a different pin: you only need to edit the one line that assigns the initial value to the variable.

Often though, the value of a variable will change while the sketch runs. For example, you could store the value read from an input into a variable. There's more information in the Variables tutorial.

Functions

A function (aka “**procedure**” or “**sub-routine**”) is a named piece of code that can be used from elsewhere in a sketch. For example, here's the definition of the setup() function:

```
void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as LED output
}
```

The first line provides information about the function, like its name, "setup". The text before and after the name specify its return type and parameters: these will be explained later. The code between the **{** and **}** is called the *body* of the function (what it does).

You can *call a function* that's already been defined (either in your sketch or as part of the Arduino language). For example, the line `pinMode (ledPin, OUTPUT);` calls the pinMode() function, passing it two *parameters*: `ledPin` and `OUTPUT`. These parameters are used by the pinMode() function to decide which pin and mode to set.

pinMode(), digitalWrite(), and delay()

The pinMode() function configures a pin as either an input or an output. To use it, you pass it the number of the pin to configure and the constant `INPUT` or `OUTPUT`. When configured as an input, it can detect the state of a sensor like a pushbutton; this is discussed here. As an output, it can drive an actuator like an LED.

The digitalWrite() functions outputs a value on a pin. For example, the following line sets the `ledPin` (pin 13) to HIGH, or 5 volts. Conversely, writing a LOW to `ledPin` connects it to ground, or 0 volts.

```
digitalWrite(ledPin, HIGH);
```

The delay() causes the Arduino to wait for the specified number of milliseconds before continuing on to the next line. There are 1000 milliseconds in a second, so the following line creates a delay of one second: `delay(1000);`

setup() and loop()

There are two *special functions* that are a part of every Arduino sketch: setup() and loop().

The setup() is called once, when the sketch starts. It's a good place to do setup tasks like setting pin modes or initializing libraries. The loop() function is called over and over and is heart of most sketches. You need to include both functions in your sketch, even if you don't need them for anything.

Further Learning: setup(), loop(), pinMode(), digitalWrite(), delay()

Simple Experiments & Examples

Bare Minimum Code Needed

This example contains the bare minimum of code you need for a sketch to compile properly on Arduino Software (IDE): the setup() method and the loop() method.

Hardware Required: Only your Arduino or Genuino Board is needed for this example.

Code

The setup() function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The setup function will only run once, after each powerup or reset of the board.

After creating a setup() function, the loop() function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond as it runs. Code in the loop() section of your sketch is used to actively control the board.

The code below won't actually do anything, but it's structure is useful for copying and pasting to get you started on any sketch of your own. It also shows you how to make comments in your code.

Any line that starts with two slashes (//) will not be read by the compiler, so you can write anything you want after it. The two slashes may be put after functional code to keep comments on the same line. Commenting your code like this can be particularly helpful in explaining, both to yourself and others, how your program functions step by step.

[Get The Code Here]

For Further Study: setup() reference, loop() reference

Blink

The equivalent of “Hello world!”, is example shows the simplest thing you can do with an Arduino or Genuino to see physical output: it blinks an LED.

Hardware Required: Arduino or Genuino Board

Optional For External Circuit:

- LED
- 220Ω resistor

Circuit

This example uses the built-in LED that most Arduino and Genuino boards have. This LED is connected to a digital pin and its number may vary from board type to board type. To make your life easier, we have a constant that is specified in every board descriptor file.

This constant is *LED_BUILTIN* and allows you to control the built-in LED easily.

Here are the correspondences between the constant and the digital pin:

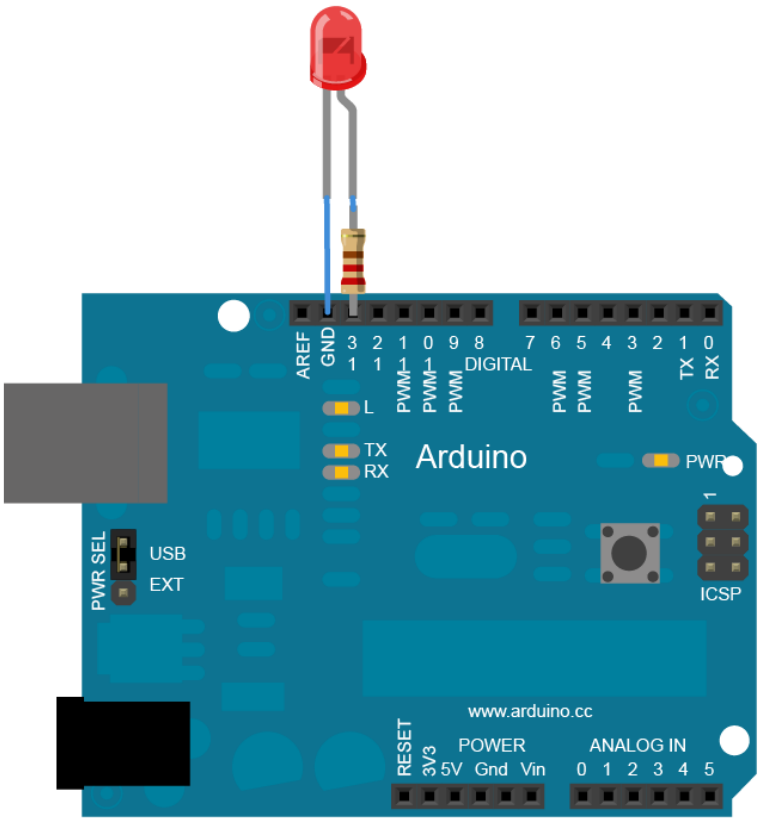
- **D1**: Gemma
- **D6**: MKR1000
- **D13**: 101, Due, Intel Edison, Intel Galileo Gen2, Leonardo and Micro, LilyPad and LilyPad USB, MEGA2560, Mini, Nano, Pro and Pro Mini, UNO, Yún, Zero

If you want to light up an **external LED** with this sketch, you need to build this circuit; where you connect one end of the resistor to the digital pin correspondent to the *LED_BUILTIN* constant.

Connect the anode of the LED (the long, positive leg) to the other end of the resistor.

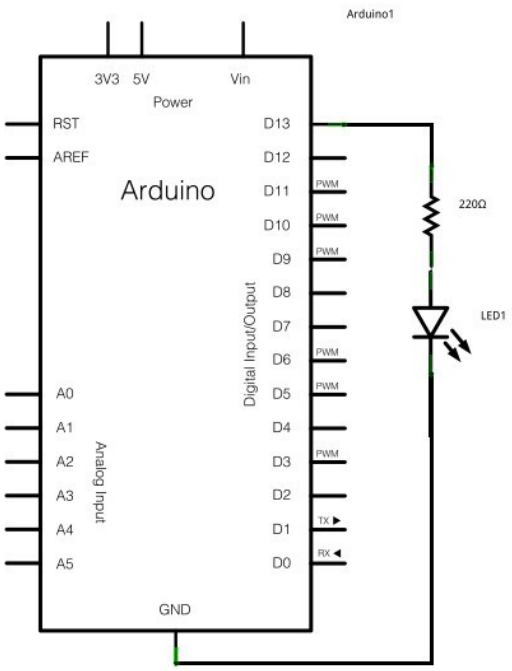
Connect the cathode of the LED (the short, negative leg) to the GND. In the diagram below we show an UNO board that has D13 as the LED_BUILTIN value.

The value of the resistor in series with the LED may be of a different value than 220Ω; the LED will lit up also with values up to 1KΩ.



This image was developed using Fritzing. For more circuit examples, see the Fritzing project page

Schematic:



Code

After you build the circuit plug your Arduino or Genuino board into your computer, start the Arduino IDE and enter the code below.

You may also load it from the menu: File/Examples/01.Basics/Blink

The first thing you do is to initialize LED_BUILTIN pin as an output pin with the line

```
pinMode(LED_BUILTIN, OUTPUT);
```

In the main loop, you turn the LED on with the line:

```
digitalWrite(LED_BUILTIN, HIGH);
```

This supplies 5 volts to the LED anode. That creates a voltage difference across the pins of the LED, and lights it up. Then you turn it off with the line:

```
digitalWrite(LED_BUILTIN, LOW);
```

That takes the LED_BUILTIN pin back to 0 volts, and turns the LED off. In between the on and the off, you want enough time for a person to see the change, so the delay() commands tell the board to do nothing for 1000 milliseconds, or one second.

When you use the delay() command, nothing else happens for that amount of time. Once you've understood the basic examples, check out the BlinkWithoutDelay example to learn how to create a delay while doing other things.

Once you've understood this example, check out the DigitalReadSerial example to learn how read a switch connected to the board.

[Get Complete Code Here]

For Further Study: setup(), loop(), pinMode(), digitalWrite(), delay()

Analog Read Serial

This example shows you how to read analog input from the physical world using a potentiometer (aka **"pot**"). A pot is a simple mechanical device that provides a varying amount of resistance when its shaft is turned.

By passing voltage through a pot and into an analog input on your board, it is possible to measure the amount of resistance produced by a pot as an analog value.

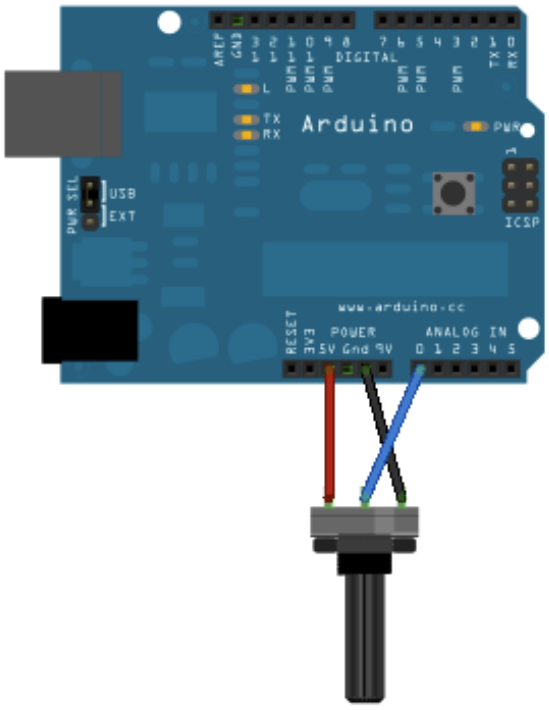
In this example you will monitor the state of your pot after establishing serial communication between your Arduino/Genuino and your computer via the Arduino IDE.

Hardware Required:

- Arduino or Genuino Board
- 10kΩ pot

Circuit

Connect the three wires from the pot to your board. The first goes from one of the outer pins of the pot to ground . The second goes from the other outer pin of the pot to 5 volts. The third goes from the middle pin of the pot to the analog pin A0.



This image was developed using Fritzing. For more circuit examples, see the Fritzing project page

By turning the shaft of the pot, you change the amount of resistance on either side of the wiper, which is connected to the center pin of the pot. This changes the voltage at the center pin.

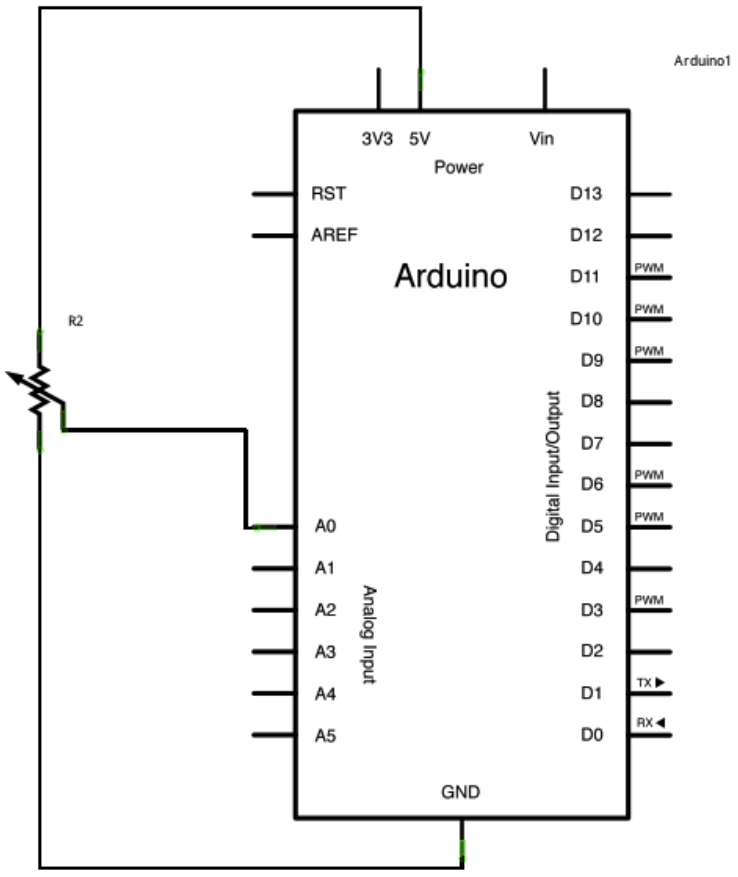
When the resistance between the center and the side connected to 5 volts is close to zero (and the resistance on the other side is close to 10kΩ), the voltage at the center pin nears 5 volts.

When the resistances are reversed, the voltage at the center pin nears 0 volts, or ground. This voltage is the *analog voltage* that you're reading as an input.

The Arduino and Genuino boards have a circuit inside called an *analog-to-digital converter* (**ADC**) that reads this changing voltage and converts it to a number between 0 and 1023.

When the shaft is turned all the way in one direction, there are 0 volts going to the pin, and the input value is 0. When the shaft is turned all the way in the opposite direction, there are 5 volts going to the pin and the input value is 1023. In between, analogRead() returns a number between 0 and 1023; proportional to the voltage being applied to the pin.

Schematic:



The Code

In the sketch below, the only thing that you do in the setup function is to begin serial communications, at 9600 bits of data per second, between your board and your computer with the command:

```
Serial.begin(9600);
```

Next, in the main loop of your code, you need to establish a variable to store the resistance value (which will be between 0 and 1023, perfect for an int datatype) coming in from your pot:

```
int sensorValue = analogRead(A0);
```

Finally, you need to print this information to your serial monitor window. You can do this with the command Serial.println() in your last line of code:

```
Serial.println(sensorValue)
```

Now, open your Serial Monitor in the Arduino IDE by clicking the icon that looks like a lens, on the right, in the green top bar (or using the keyboard shortcut **Ctrl+Shift+M**). You should see a steady stream of numbers ranging from 0-1023, correlating to the position of the pot.

As you turn your pot, these numbers will respond almost instantly.

[Get The Code Here]

For Further Study: setup(), loop(), analogRead(), int, serial

Digital Read Serial

This example shows you how to monitor the state of a switch by establishing serial communication between your Arduino or Genuino and your computer over USB.

Hardware Required:

- Arduino/Genuino Board
- A momentary switch (i.e. button, toggle, etc.)
- 10kΩ resistor
- Hook-up wires
- Breadboard

Circuit

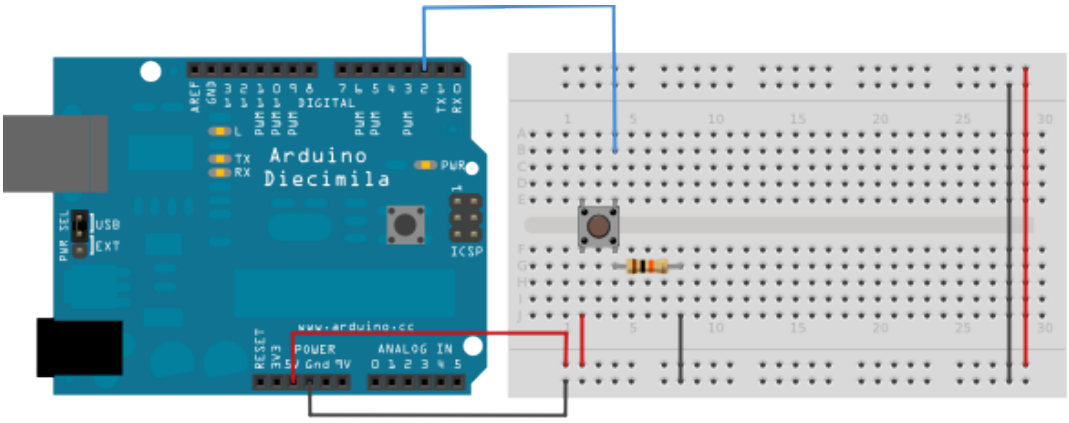


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Connect three wires to the board. The first two, red and black, connect to the two long vertical rows on the side of the breadboard to provide access to the 5 volt supply and ground.

The third wire goes from digital pin 2 to one leg of the pushbutton. That same leg of the button connects through a pull-down resistor (10kΩ) to ground. The other leg of the button connects to the 5 volt supply.

Pushbuttons or switches connect two points in a circuit when you press them.

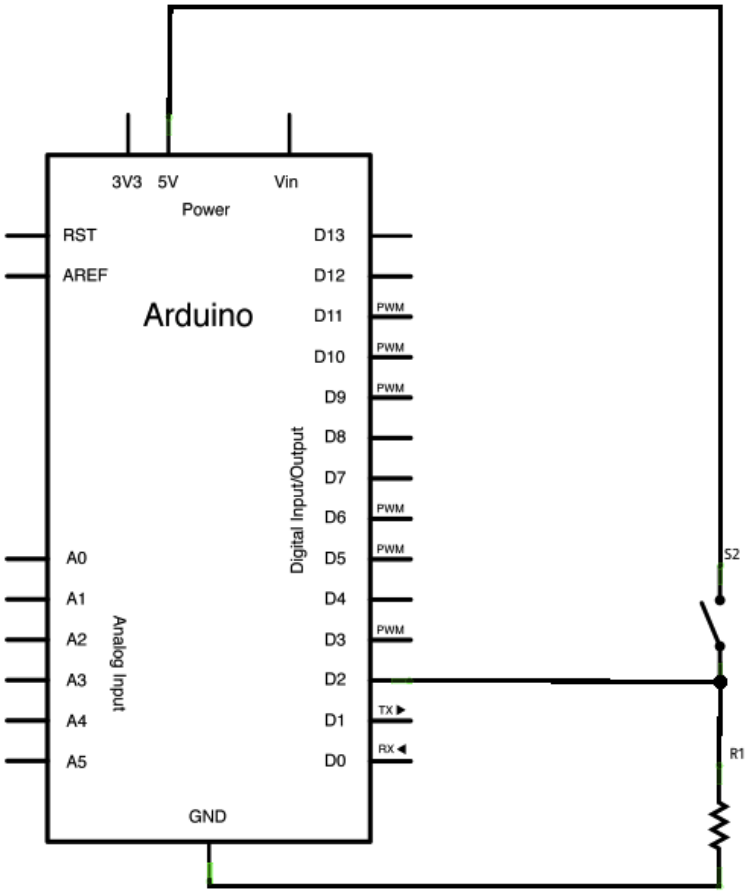
When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and reads as LOW, or 0.

When the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that the pin reads as HIGH, or 1.

If you disconnect the digital i/o pin from everything, the LED may blink erratically. This is because the input is "floating"; that is, it doesn't have a solid connection to voltage or ground, and it will randomly return either HIGH or LOW.

That's why you need a pull-down resistor in the circuit.

Schematic:



Code

The very first thing that you do will in the setup function is to begin serial communications, at 9600 bits of data per second, between your board and your computer with the line:

```
Serial.begin(9600);
```

Next, initialize digital pin 2, the pin that will read the output from your button, as an input:

```
pinMode(2,INPUT);
```

Now that your setup has been completed, move into the main loop of your code. When your button is pressed, 5 volts will freely flow through your circuit, and when it is not pressed, the input pin will be connected to ground through the 10kΩ resistor.

This is a digital input, meaning that the switch can only be in either an on state (seen by your Arduino as a "1", or HIGH) or an off state (seen by your Arduino as a "0", or LOW), with nothing in between.

The first thing you need to do in the main loop of your program is to establish a variable to hold the information coming in from your switch.

Since the information coming in from the switch will be either a "1" or a "0", you can use an int datatype. The variable **sensorValue** is set to what is being read on D2. You can accomplish all this with just one line of code:

```
int sensorValue = digitalRead(2);
```

Once the bord has read the input, make it print this information back to the computer as a decimal value. You can do this with the command Serial.println() in our last line of code:

```
Serial.println(sensorValue);
```

Now, when you open your Serial Monitor in the Arduino Software (IDE), you will see a stream of "0"s if your switch is open, or "1"s if your switch is closed.

[Get The Code Here]

For Further Study: setup(), loop(), pinMode(), digitalRead(), delay(), int, serial, DigitalPins

Fade

This example demonstrates the use of the analogWrite() function in fading an LED off and on. AnalogWrite uses pulse width modulation (PWM), turning a digital pin on and off very quickly with different ratio between on and off, to create a fading effect.

Hardware Required:

- Arduino or Genuino board
- LED
- 220Ω resistor
- Hook-up wires
- Breadboard

Circuit

Connect the anode (the longer, positive leg) of your LED to digital output pin 9 on your board through a 220Ω resistor. Connect the cathode (the shorter, negative leg) directly to ground.

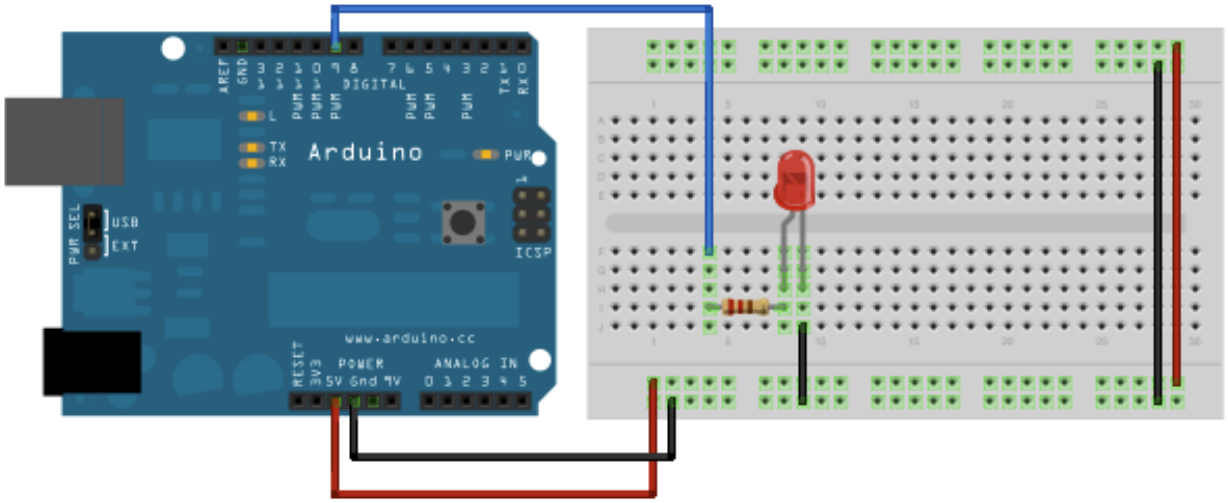
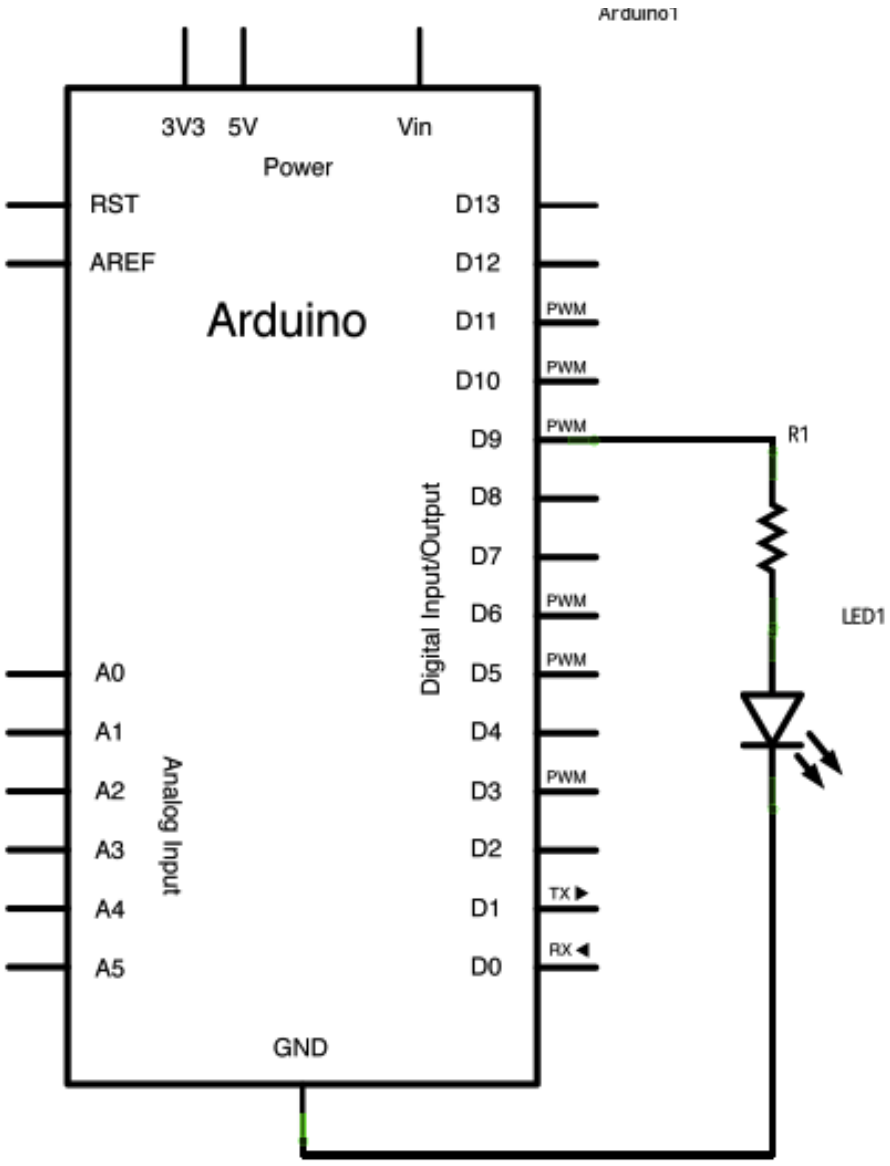


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Schematic:



Code

After declaring pin 9 to be your ledPin, there is nothing to do in the setup() function of your code.

The analogWrite() function that you will be using in the main loop of your code requires two arguments: One telling the function which pin to write to, and one indicating what PWM value to write.

In order to fade your LED off and on, gradually increase your PWM value from 0 (all the way off) to 255 (all the way on), and then back to 0 once again to complete the cycle.

In the corresponding sketch, the PWM value is set using a variable called `brightness`. Each time through the loop, it increases by the value of the variable fadeAmount.

If `brightness` is at either extreme of its value (either 0 or 255), then `fadeAmount` is changed to its negative:

- So if `fadeAmount` is 5, then `brightness` is set to -5.
- Conversely, if `fadeAmount` is -5, then it's set to 5.

The next time through the loop, this change causes `brightness` to change direction as well.

AnalogWrite() can change the PWM value very fast, so the delay at the end of the sketch controls the speed of the fade. Try changing the value of the delay and see how it changes the fading effect.

[Get The Code Here]

For Further Study: setup(), loop(), analogWrite(), int, for, PWM

Read Analog Voltage

Very similar to the previous example "**Analog Read Serial", t**his example shows you how to read an analog input on analog pin 0, convert the values from analogRead() *into voltage*, and print it out to the serial monitor of the Arduino IDE.

Hardware Required:

- Arduino or Genuino Board
- 10kΩ pot

Circuit

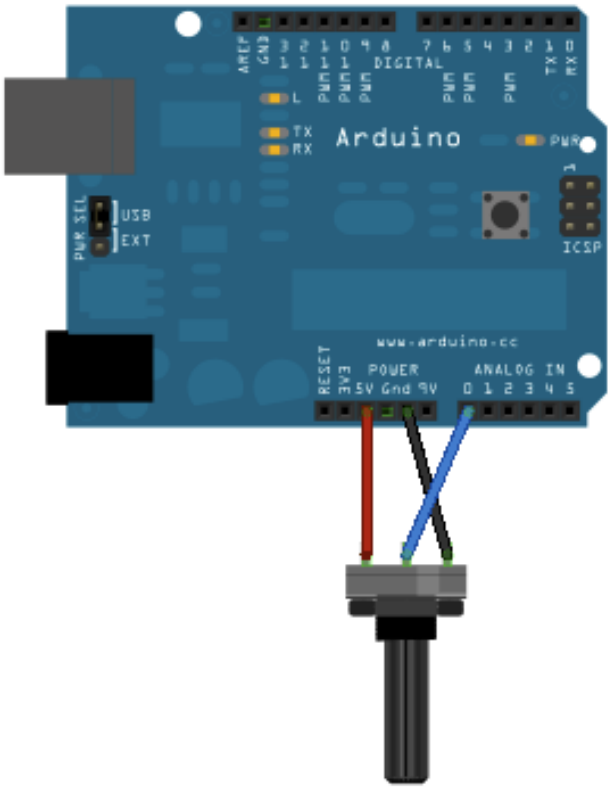


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Connect the three wires from the pot to your board. The first goes to ground from one of the outer pins of the pot. The second goes to 5 volts from the other outer pin of the pot. The third goes from the middle pin of the potentiometer to analog input 2.

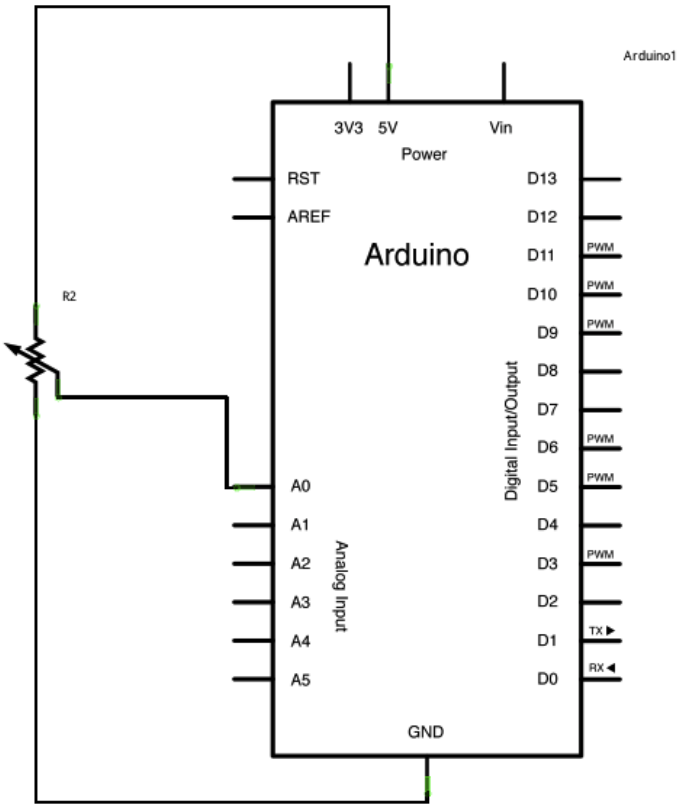
By turning the shaft of the pot, you change the amount of resistance on either side of the wiper which is connected to the center pin of the pot. This changes the center pin voltage.

When the resistance between the center and the side connected to 5 volts is close to zero (and the resistance on the other side is close to 10kΩ), the voltage at the center pin nears 5 volts. Conversely, when the resistances are reversed, the voltage at the center pin nears 0 volts, or ground. This voltage is the **analog voltage** that you're reading as an input.

As you may recall, he microcontroller of the board has a circuit inside called an *analog-to-digital converter* (**ADC**) that reads this changing voltage and converts it to a number between 0 and 1023:

- When the shaft is turned all the way in one direction, there are 0 volts going to the pin, and the input value is 0. Conversely, when the shaft is turned all the way in the opposite direction, there are 5 volts going to the pin and the input value is 1023.
- In between,analogRead() returns a number between 0 and 1023 that is proportional to the amount of voltage being applied to the pin.

Schematic:



Code

In the program below, the very first thing that you do will in the setup function is to begin serial communications, at 9600 bits of data per second, between your board and your computer with the line:

```
Serial.begin(9600);
```

Next, in the main loop of your code, you need to establish a variable to store the resistance value (which will be between 0 and 1023, perfect for an int datatype) coming in from your potentiometer:

```
int sensorValue = analogRead(A0);
```

To change the values from 0-1023 to a range that corresponds to the voltage the pin is reading, you'll need to create another variable, a float, and do a little math. To scale the numbers between 0.0 and 5.0, divide 5.0 by 1023.0 and multiply that by **sensorValue**:

Float Voltage = sensorValue * (5.0 / 1023.0);

Finally, you need to print this information to your serial window as. You can do this with the command Serial.println() in your last line of code:

```
Serial.println(voltage)
```

Now, when you open your Serial Monitor in the Arduino IDE, you should see a steady stream of numbers ranging from 0.0 - 5.0. As you turn the pot, the values change, corresponding to the voltage coming into pin A0.

[Get The Code Here]

For Further Study: setup(), loop(), analogRead(), int, Serial, float

Blink Without Delay

Sometimes you need to do two things at once. For example you might want to blink an LED while reading a button press. In this case, you can't use delay(), because Arduino pauses your program during the delay(). If the button is pressed while Arduino is paused waiting for the delay() to pass, your program will miss the button press.

Hence this sketch turns on the LED on and then makes note of the time. Then, each time through loop(), it checks to see if the desired blink time has passed. If it has, it toggles the LED on or off and makes note of the new time. In this way the LED blinks continuously while the sketch execution never lags on a single instruction.

Hardware Required:

- Arduino or Genuino Board
- LED
- 220Ω resistor

Circuit

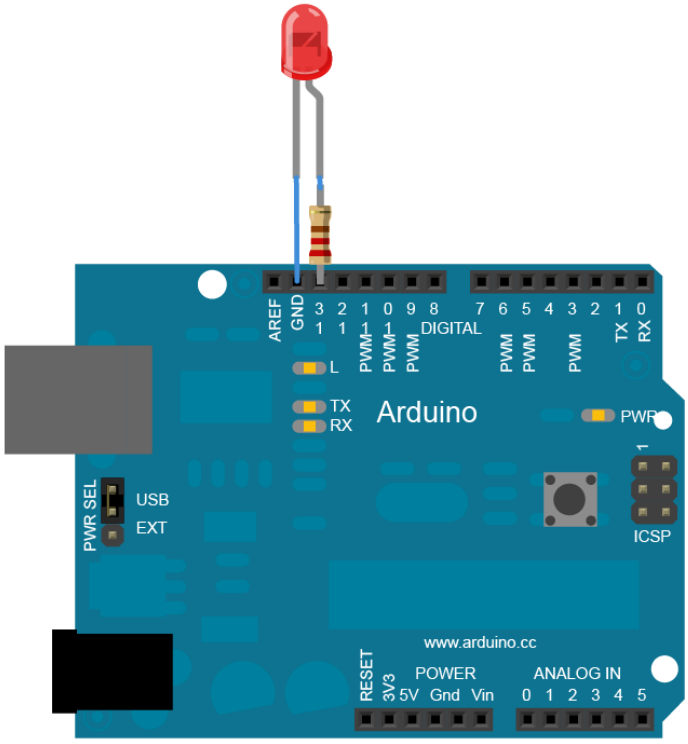
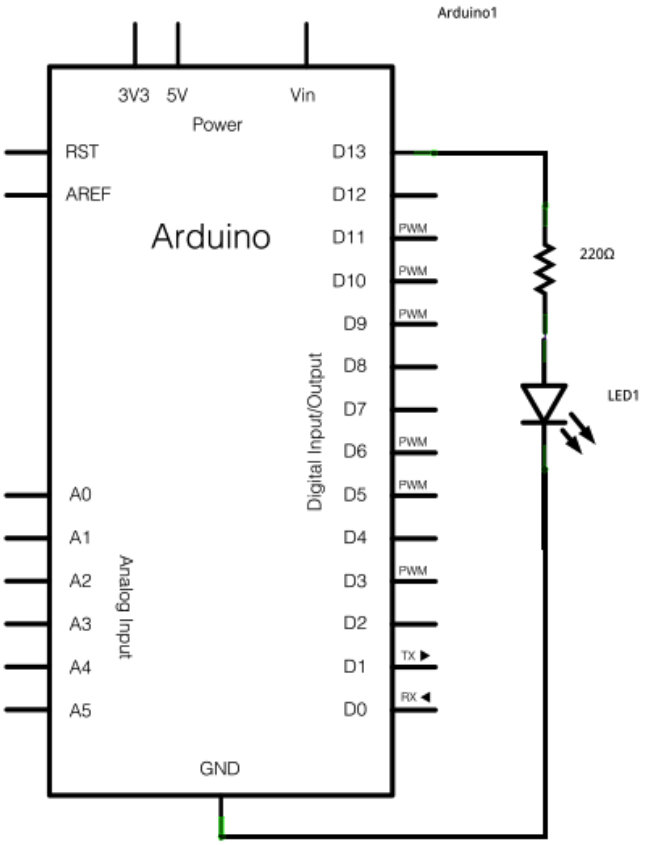


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

To build the circuit, connect one end of the resistor to pin 13 of the board. Connect the anode leg of the LED (the long, positive leg) to the other end of the resistor. Connect the cathode leg of the LED (the short, negative leg) to the board GND, as shown in the diagram above and the schematic below.

Most Arduino and Genuino boards already have an LED attached to pin 13 on the board itself. If you run this example with no hardware attached, you should see that LED blink.

Schematic:



After you build the circuit plug your board into your computer, start the Arduino IDE, and Import the code below. The code below uses the millis() function, a command that returns the number of milliseconds since the board started running its current sketch, to blink an LED.

[Get The Code Here]

For Further Study: setup(), loop(), millis()

Button

Pushbuttons or switches connect two points in a circuit when you press them. This example turns on the built-in LED on pin 13 when you press the button.

Required Hardware:

- Arduino or Genuino Board
- Momentary button or Switch
- 10KΩ resistor
- Hook-up wires
- Breadboard

Circuit

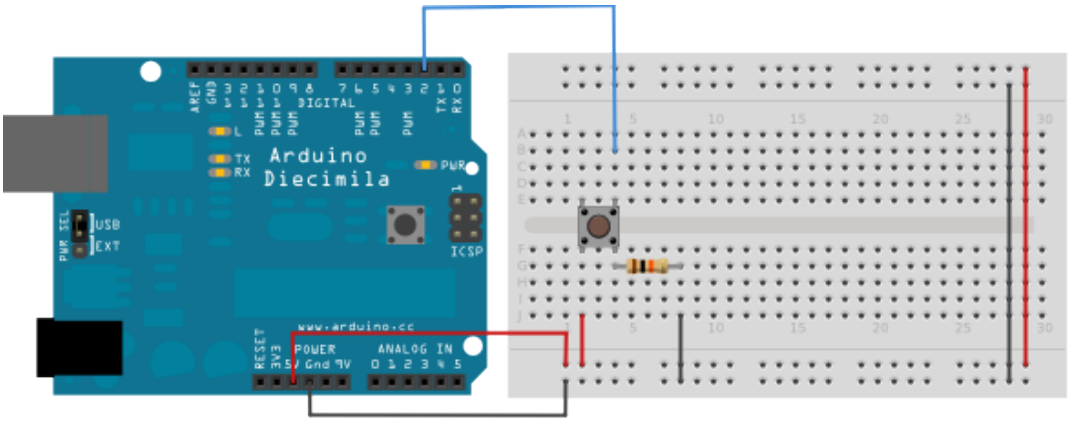


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

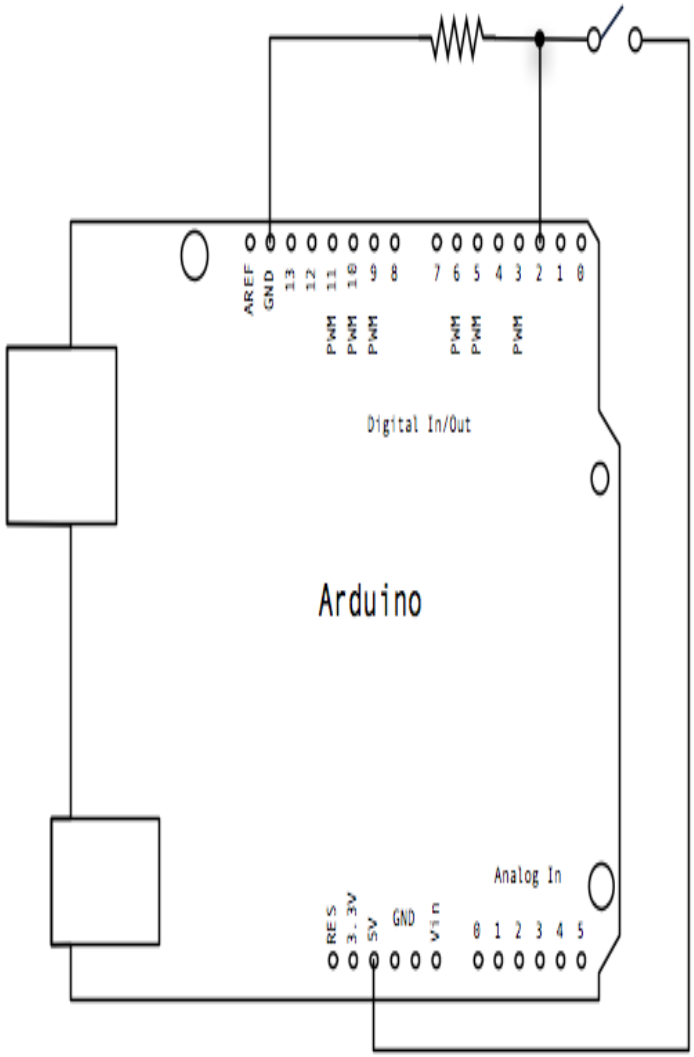
Connect three wires to the board. The first two, red and black, connect to the two long vertical rows on the side of the breadboard to provide the 5v and ground access. The third wire goes from digital pin 2 to one leg of the pushbutton. That same leg of the button connects through a 10KΩ pull-down resistor to ground. The other leg of the button connects to the 5 volt supply.

When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and we read a LOW. When the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that we read a HIGH. Conversely, a **pullup resistor**

keeps the input HIGH, and goes LOW only when the button is pressed; hence the behavior of the sketch will be reversed (LED normally on, turning off when you press the button).

If you disconnect the digital I/O pin from everything, the LED may blink randomly between HIGH and LOW; hence the pull-up or pull-down resistor in the circuit.

Schematic:



[Get The Code Here]

For Further Study: pinMode(), digitalWrite(), digitalRead(), if, else

Debounce

Due to mechanical and physical issues, switches can be quite “noisy”; whereby misread as multiple presses in a very short time, and fooling the program. This example demonstrates how to **debounce** an input; which means checking twice in a short period of time to make sure the pushbutton is definitely pressed.

Without debouncing, pressing the button once may cause unpredictable results. This sketch uses the millis() function to keep track of the time passed since the button was pressed.

Hardware Required:

- Arduino or Genuino Board
- Momentary button or switch
- 10kΩ resistor
- Hook-up wires
- Breadboard

Circuit

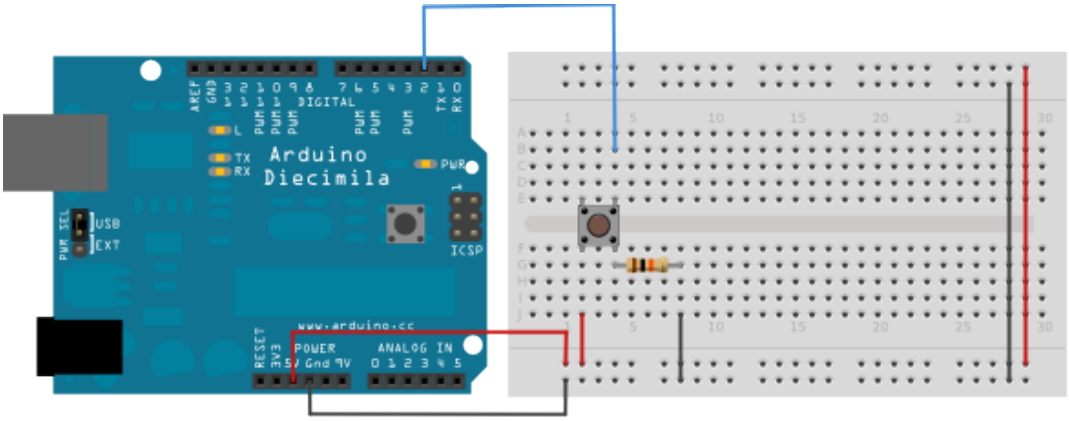
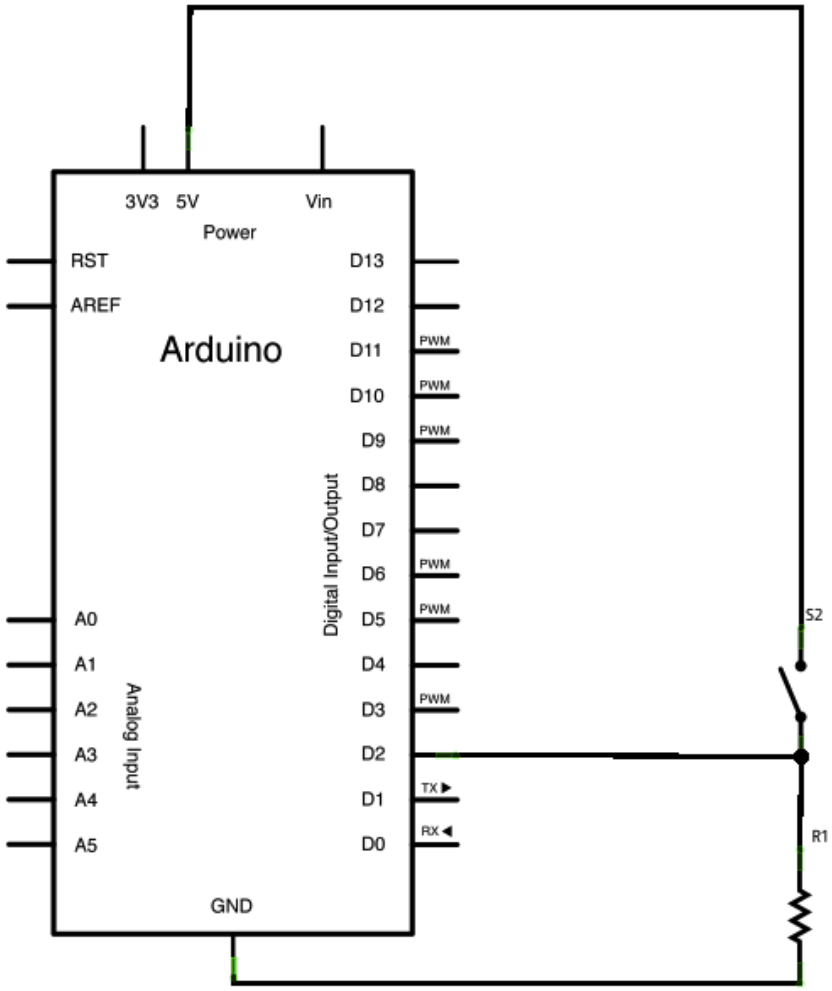


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Schematic:



Code

The sketch below is based on **Limor Fried's version of debounce**, but the *logic is inverted* from her example. In her example, the switch returns LOW when closed, and HIGH when open. Here, the switch returns HIGH when pressed and LOW when not pressed.

[Get The Code Here]

For Further Study: pinMode(), digitalWrite(), digitalRead(), if(), millis()

Input Pullup Serial

This example demonstrates the use of INPUT_PULLUP with pinMode(). It monitors the state of a switch by establishing serial communication between your Arduino and your computer over USB. Additionally, when the input is HIGH, the onboard LED attached to pin 13 will turn on; when LOW, the LED will turn off.

Hardware Required:

- Arduino Board
- Momentary button or toggle switch
- Breadboard
- Hook-up wire

Circuit

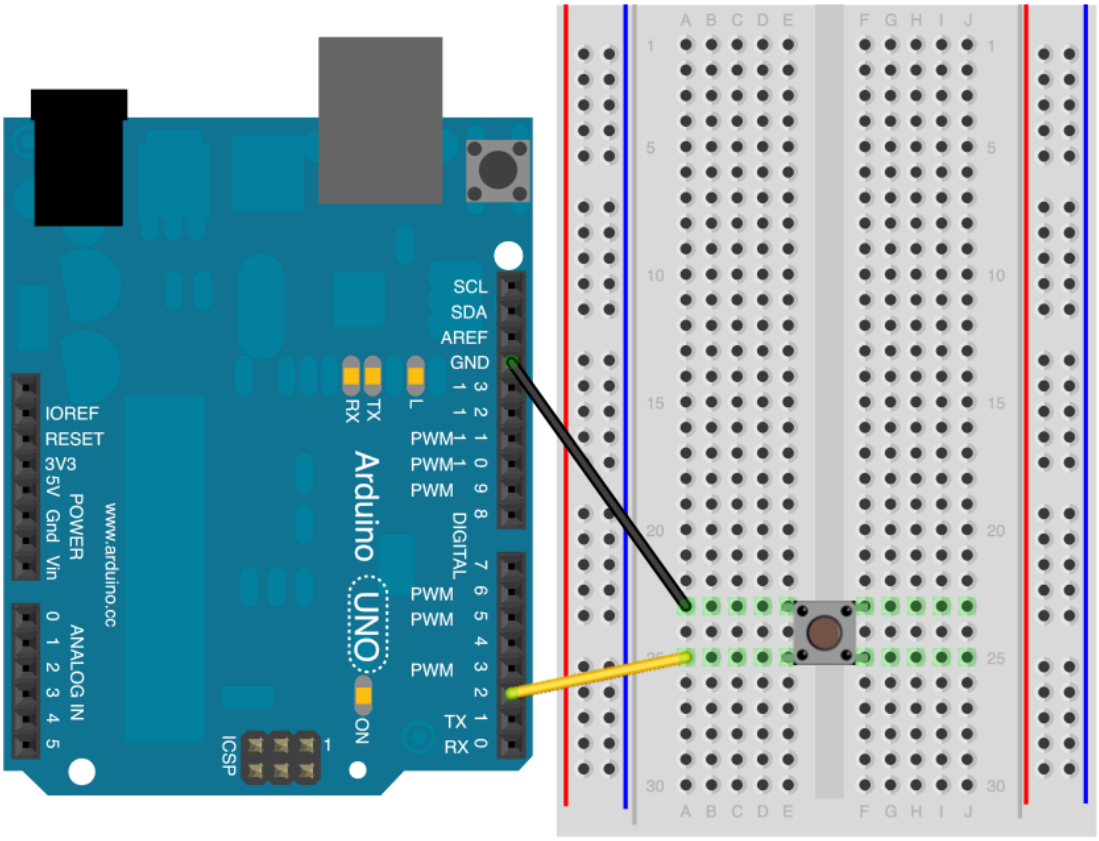


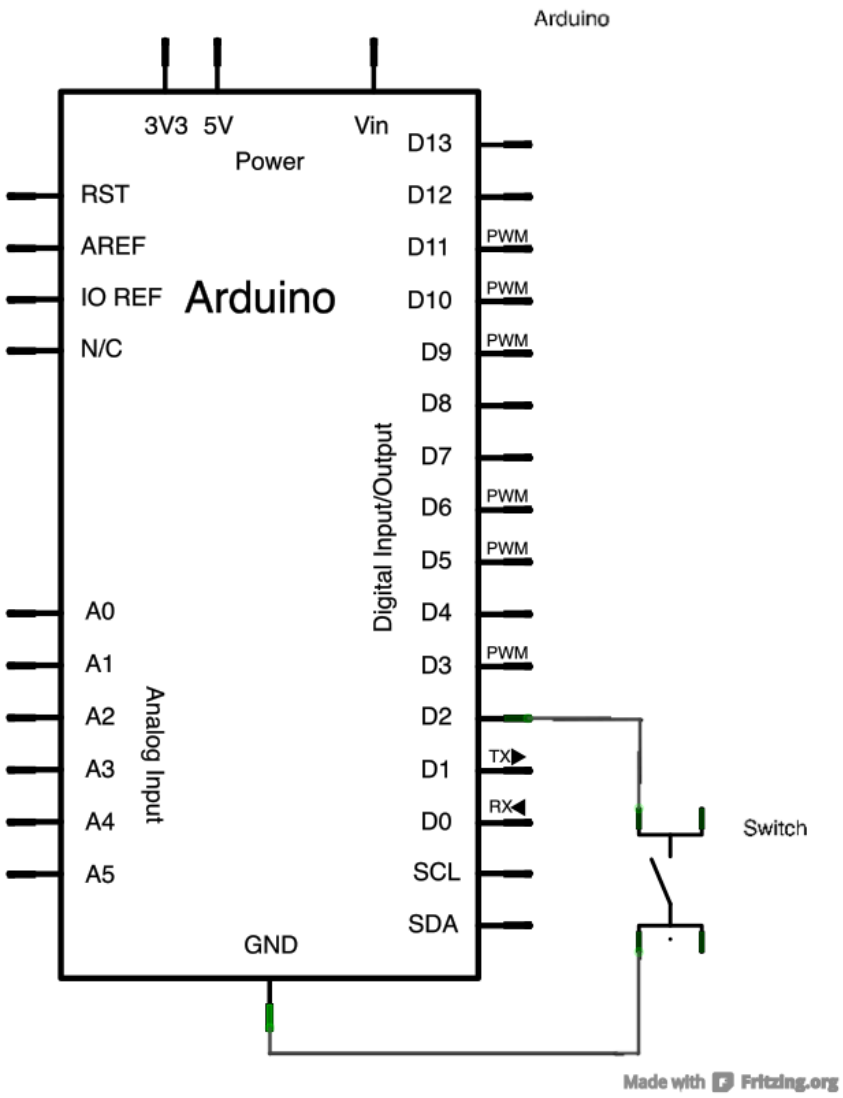
Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Connect two wires to the Arduino board. The black wire connects ground to one leg of the pushbutton. The second wire goes from digital pin 2 to the other leg of the pushbutton.

Pushbuttons or switches connect two points in a circuit when you press them. When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton.

Because the internal pull-up on pin 2 is active and connected to 5V, we read HIGH when the button is open. When the button is closed, the Arduino reads LOW because a connection to ground is completed.

Schematic:



Code

In this sketch, the very first thing that you do will in the setup function is to begin serial communications, at 9600 bits of data per second, between your Arduino and your computer with the line:

```
Serial.begin(9600);
```

Next, initialize digital pin 2 as an input with the internal pull-up resistor enabled:

```
pinMode(2,INPUT_PULLUP);
```

The following line make pin 13, with the onboard LED, an output:

```
pinMode(13, OUTPUT);
```

Now that your setup has been completed, move into the main loop of your code. When your button is not pressed, the internal pull-up resistor connects to 5 volts. This causes the Arduino to report "1" or HIGH. When the button is pressed, the Arduino pin is pulled to ground, causing the Arduino report a "0", or LOW.

The first thing you need to do in the main loop of your program is to establish a variable to hold the information coming in from your switch. Since the information coming in from the switch will be either a "1" or a "0", you can use an int datatype.

Call this variable sensorValue, and set it to equal whatever is being read on digital pin 2. You can accomplish all this with just one line of code:

```
int sensorValue = digitalRead(2);
```

Once the Arduino has read the input, make it print this information back to the computer as a decimal (DEC) value. You can do this with the command Serial.println() in our last line of code:

```
Serial.println(sensorValue, DEC);
```

Now, when you open your Serial Monitor in the Arduino environment, you will see a stream of "0"s if your switch is closed, or "1"s if your switch is open. The LED on pin 13 will illuminate when the switch is HIGH, and turn off when LOW.

[Get The Code Here]

Further Learning: setup(), loop(), pinMode(), digitalRead(), delay(), int, serial, DigitalPins

State Change/Edge Detection For Pushbuttons

Once you've got a pushbutton working, you often want to do some action based on how many times the button is pushed. To do this, you need to know when the button changes state from off to on, and count how many times this change of state happens. This is called **state change detection** or **edge detection.**

In this tutorial learn how to check the state change, send a message to the Serial Monitor with the relevant information, and count four state changes to turn on and off an LED.

Hardware Required:

- Arduino or Genuino Board
- Momentary button switch
- 10kΩ resistor
- Hook-up wires
- Breadboard

Circuit

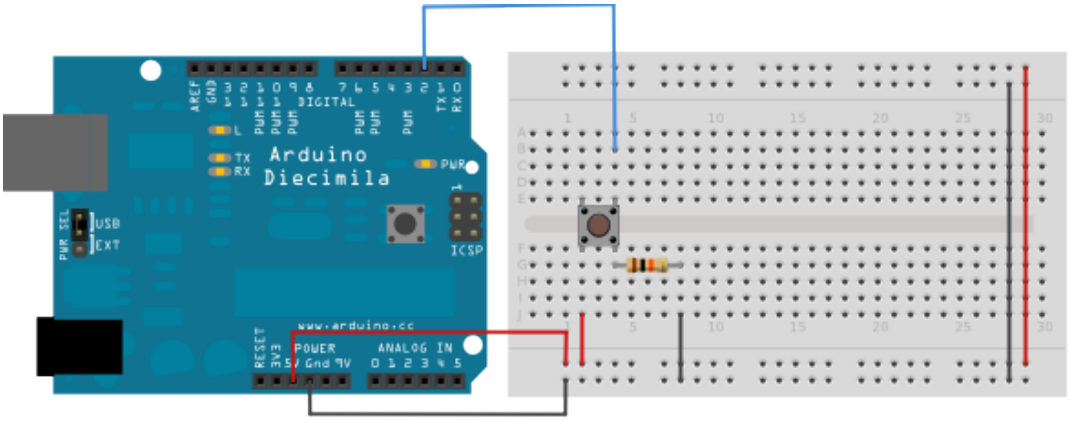
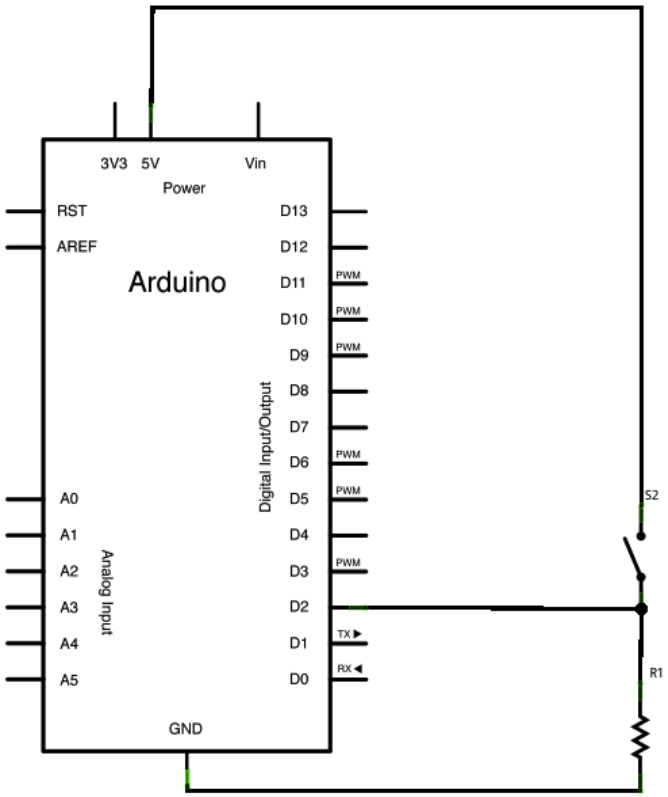


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Connect three wires to the board. The 1st goes from one leg of the pushbutton through a pull-down resistor (10kΩ) to ground. The 2nd goes from the corresponding leg of the push-button to the 5 volt supply. The 3rd connects to a digital I/O pin (pin 2) which reads the button's state. When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton. The pin is connected to ground via the pull-down resistor and we read a LOW.

Conversely, when the button is closed (pressed), it makes a connection between its two legs, connecting the pin to voltage; so that we read a HIGH (the pin is still connected to ground, but the resistor resists the flow of current, so the path of least resistance is to +5v.)

Schematic:



Code

This sketch continually reads the button's state and compares the button's state to its state the last time through the main loop. If the current button state is different from the last button state and HIGH, then the button changed from off to on; incrementing a button push counter. It also checks the button push counter's value. If it's an even multiple of four, it turns the LED on pin 13 ON. Otherwise, it turns it off.

[Get The Code Here]

Further Learning: pinMode(), digitalWrite(), digitalRead(), millis(), if

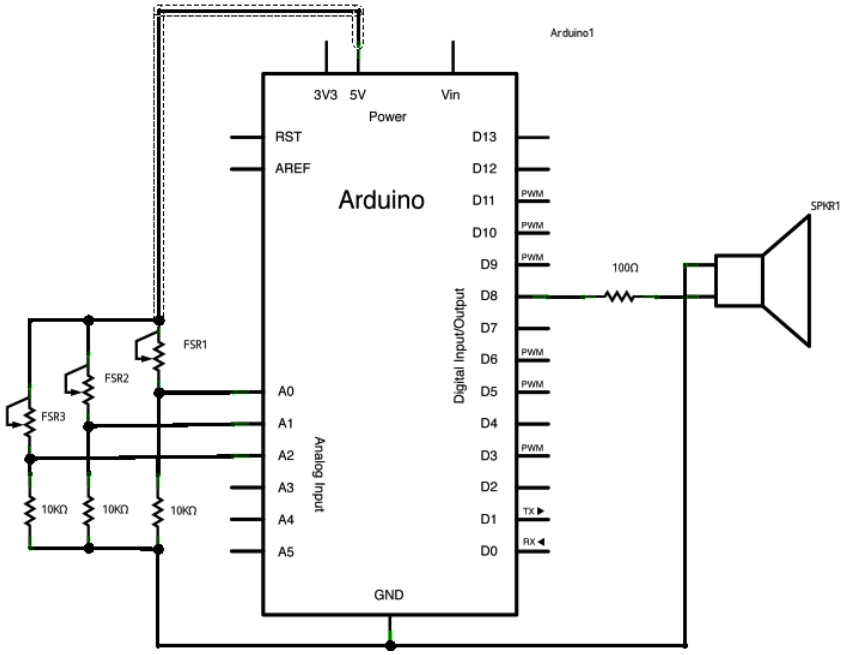
Simple 3-Tone Keyboard Using Analog Sensors

This example shows how to use the tone() command to generate different pitches depending on which sensor is pressed.

Hardware Required:

- Arduino or Genuino Board
- 8Ω speaker
- 3 force-sensing resistors or analog sensors
- 3 10kΩ resistors
- 100Ω resistor
- Hook-up wires
- Breadboard

Schematic:



Circuit

Connect one terminal of your speaker to digital pin 8 through a 100 ohm resistor, and its other terminal to ground. Power your three FSR's (or any other analog sensor) with 5V in parallel. Connect each sensor to analog pins 0-2, using a 10K resistor as a reference to ground on each input line.

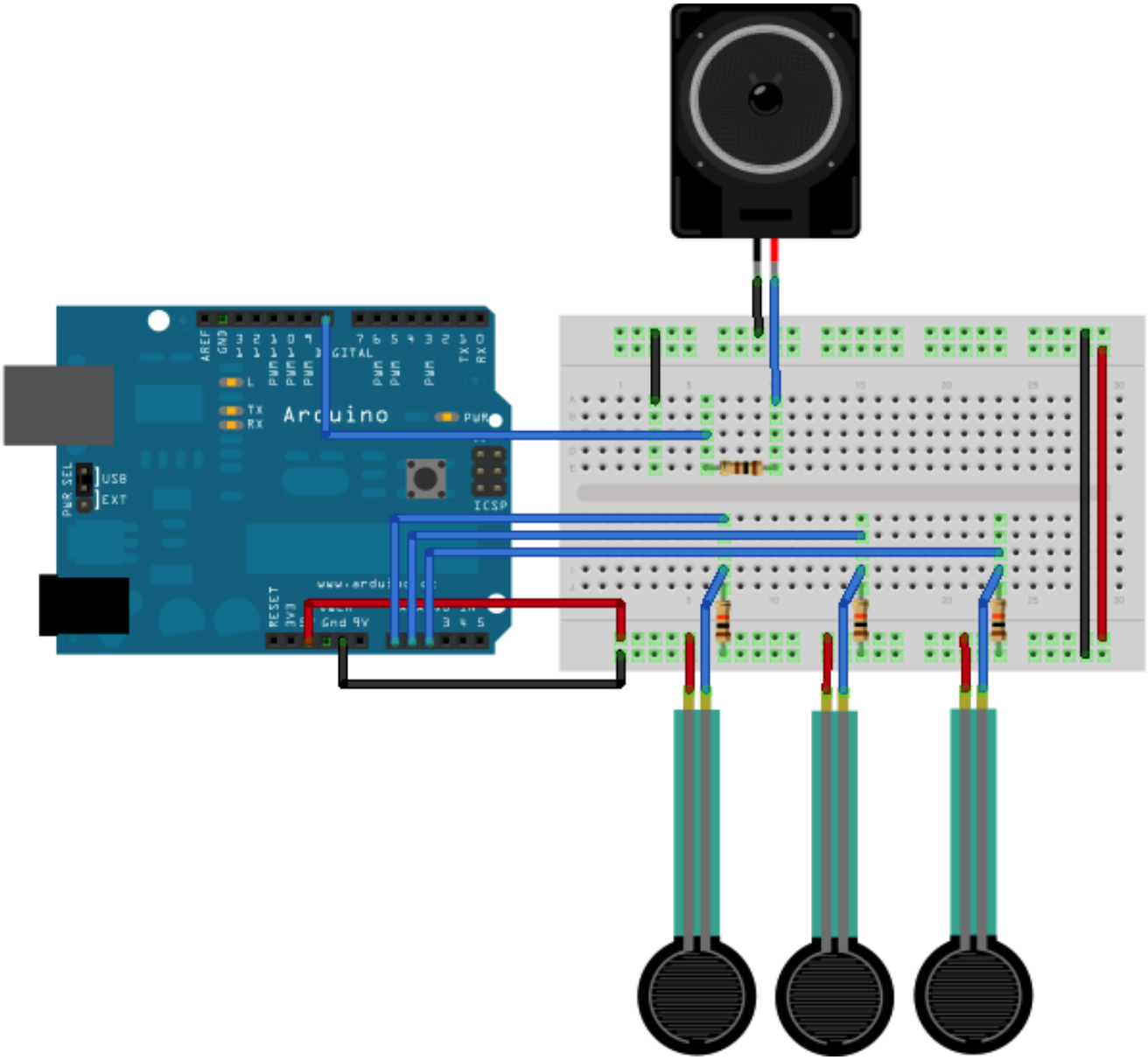


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

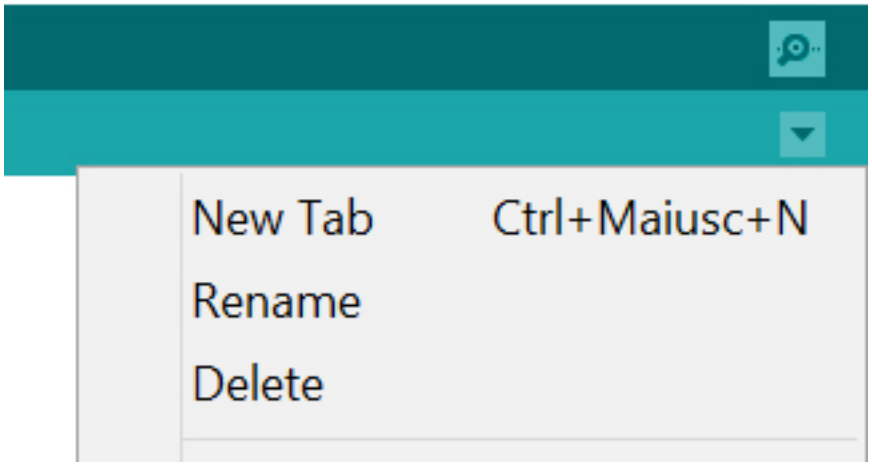
Code

The sketch below reads three analog sensors. Each corresponds to a note value in an array of notes. If any of the sensors is above a given threshold, the corresponding note is played. Here's the **main sketch**:

[Get The Code Here]

The sketch uses an **addon sketch** (pitches.h) which contains all the pitch values for typical notes. For example, NOTE_C4 is middle C. NOTE_FS4 is F sharp, and so forth. This note table was originally written by Brett Hagman, on whose work the tone() command was based.

You may find it useful for whenever you want to make musical notes. To make the pitches.h file, either click on the button just below the serial monitor icon and choose "New Tab" (or **Ctrl+Shift+N**).



Then paste in the following code:

[Get The Code Here]

Further Learning: array(), for(), tone()

Play A Melody Using The tone() Function

This example shows how to use the tone() command to generate notes. It plays a little melody you may have heard before.

Hardware Required:

- Arduino or Genuino board
- Piezo buzzer or a speaker
- Hook-up wires

Circuit

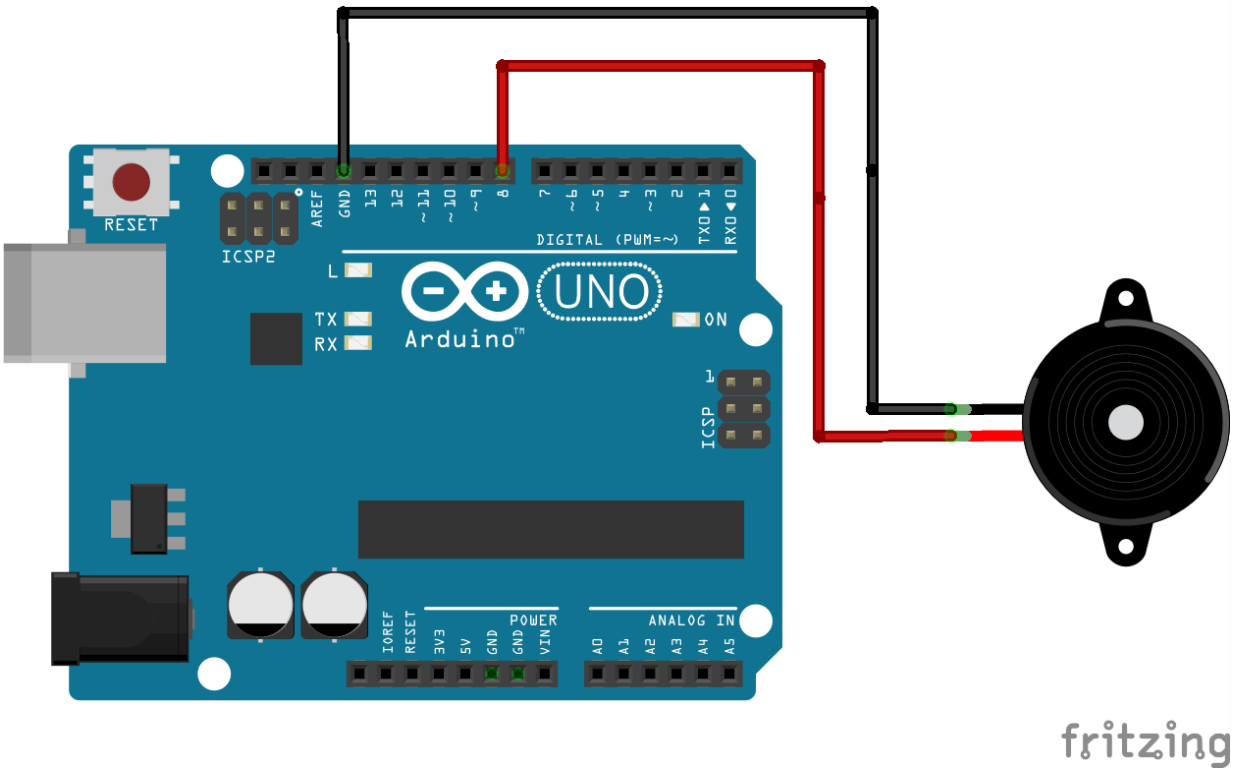
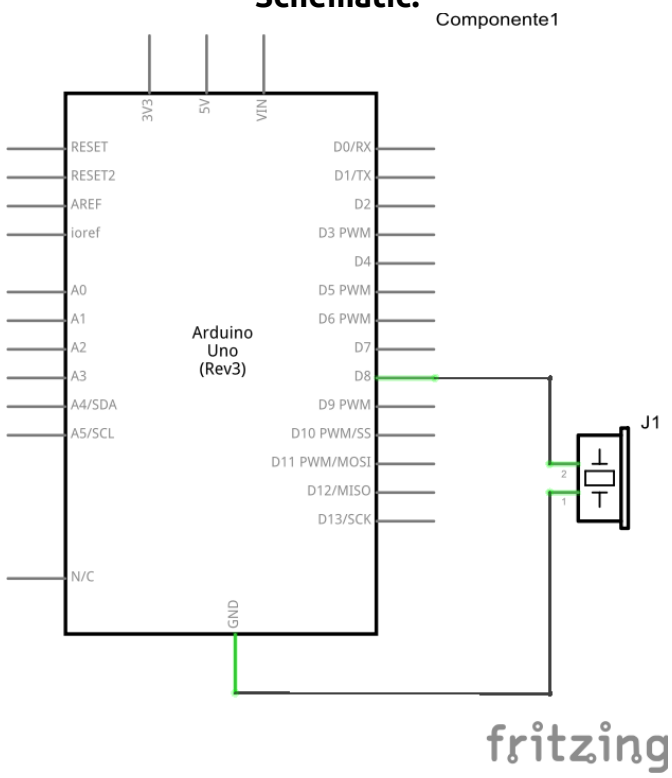


Image developed using Fritzing. For more circuit examples, see the Fritzing project page

Schematic:

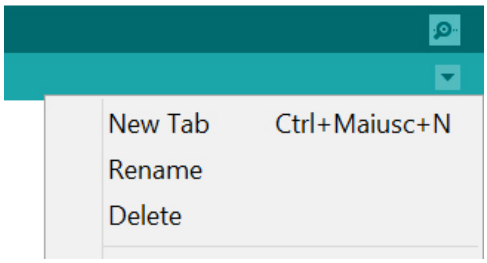


Code

[Get The Code Here]

As with the previous example, this sketch also utilizes the same extra **pitches.h** file, originally written by Brett Hagman (which contains all the pitch values for typical notes - NOTE_C4 is middle C. NOTE_FS4 is F sharp, and so forth).

To make the pitches.h file, either click on the button just below the serial monitor icon and choose "New Tab" (or **Ctrl+Shift+N**).



Then paste in the following code and save it as pitches.h:

[Get The Code Here]

Further Learning: Array(), for(), tone()

Digital Electronics Essentials

Digital Numeration Basics

In the binary numeration system, each weight constant is twice as much as the one before it. The two allowable cipher symbols for the binary system of numeration are “1” and “0,” and are arranged right-to-left in doubling values of weight.

The rightmost place is the ones place, just as with decimal notation. Proceeding to the left, we have the twos place, the fours place, the eights place, the sixteens place, and so on. For example, the following binary number can be expressed, just like the decimal number 26, as a sum of each cipher value times its respective weight constant:

11010 = 2 + 8 + 16 = 26
11010 = (1 x 16) + (1 x 8) + (0 x 4) + (1 x 2) + (0 x 1)

In the above example, we’re mixing two different kinds of numerical notation. To avoid unnecessary confusion, we have to denote which form of numeration we’re using. Typically, this is done in subscript form, with a “2” for binary and a “10” for decimal; so binary 11010_2 is equal to decimal 26_{10}.

The subscripts are not “mathematical operation” symbols like superscripts (exponents) are. They simply indicate what numeration system we’re using when writing these symbols for others to read.

- Hence, if you see “3_{10}”, it’s the **number 3** (using decimal numeration).
- Conversely, if you see “3^{10}”, this means three *to the tenth power* (59,049). If no subscript is shown, a decimal number is assumed.

Binary is referred to as “base two” numeration, and decimal as “base ten.” Additionally, we refer to each cipher position in binary as a **bit** rather than the familiar word ‘digit’ used in the decimal system.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-1/systems-of-numeration/

To convert a number in binary numeration to its equivalent in decimal form, all you have to do is calculate the sum of all the products of bits with their respective place-weight constants. To illustrate:

Convert 11001101_2 to decimal form:

bits =	1	1	0	0	1	1	0	1
.	-	-	-	-	-	-	-	-
weight =	1	6	3	1	8	4	2	1
(in decimal notation)	2	4	2	6				
	8							

The bit on the *far right side* is called the **Least Significant Bit (LSB)**, because it stands in the place of the lowest weight (the one's place). The bit on the *far left side* is called the **Most Significant Bit (MSB)**, because it stands in the place of the highest weight (the 128's place).

Remember, a bit value of "1" means that the respective place weight gets added to the total value, and a bit value of "0" means that the respective place weight does not get added to the total value. With the above example, we have:

$128_{10} + 64_{10} + 8_{10} + 4_{10} + 1_{10} = 205_{10}$

If we encounter a binary number with a dot (.), called a "**binary point**" instead of a decimal point, we follow the same procedure; each place weight to the right of the point is *one-half the value of the one to the left of it* (just as each place weight to the right of a decimal point is one-tenth the weight of the one to the left of it). For example:

Convert 101.011_2 to decimal form:

.							
bits =	1	0	1	.	0	1	1
.	-	-	-	-	-	-	-
weight =	4	2	1		1	1	1
(in decimal					/	/	/
notation)					2	4	8

$4_{10} + 1_{10} + 0.25_{10} + 0.125_{10} = 5.375_{10}$

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-1/decimal-versus-binary-numeration/

Octal is a base-of-eight digital numeration system. Valid ciphers include the symbols 0, 1, 2, 3, 4, 5, 6, and 7. Each place weight differs from the one next to it by a factor of eight.

Hexadecimal is a base-of-sixteen digital numeration system. Valid ciphers include the normal decimal symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus six alphabetical characters A, B, C, D, E, and F; for a total of sixteen. As you likely guessed already, each place weight differs by a factor of sixteen.

To put all of this into perspective, the following chart shows zero to twenty using decimal, binary, octal, and hexadecimal to contrast these systems of numeration:

Number	Decimal	Binary	Octal	Hexadecimal
Zero	0	0	0	0
One	1	1	1	1
Two	2	10	2	2
Three	3	11	3	3
Four	4	100	4	4
Five	5	101	5	5
Six	6	110	6	6
Seven	7	111	7	7
Eight	8	1000	10	8
Nine	9	1001	11	9
Ten	10	1010	12	A
Eleven	11	1011	13	B
Twelve	12	1100	14	C
Thirteen	13	1101	15	D
Fourteen	14	1110	16	E
Fifteen	15	1111	17	F
Sixteen	16	10000	20	10
Seventeen	17	10001	21	11
Eighteen	18	10010	22	12
Nineteen	19	10011	23	13
Twenty	20	10100	24	14

Octal and hexadecimal numeration systems would be pointless if not for their ability to be *easily converted to and from binary notation*. In essence, they serve as a more human-friendly "shorthand" way of denoting a binary number.

Because the bases of octal (eight) and hexadecimal (sixteen) are even multiples of binary's base (two), binary bits can be grouped together and directly converted to/from their respective octal or hexadecimal counterparts. With octal, the binary bits are grouped in three's (because 23 = 8), and with hexadecimal, the binary bits are grouped in four's (because 24 = 16).

```
BINARY TO OCTAL CONVERSION
Convert 10110111.1₂  to octal:
.
.                          implied zero       implied zeros
.                               |                  ||
.                              010   110   111   100
Convert each group of bits     ###   ###   ### . ###
to its octal equivalent:        2     6     7     4
.
Answer:     10110111.1₂  =  267.4₈
```

We had to group the bits in three's, from the binary point left, and from the binary point right, adding (implied) zeros as necessary to make complete 3-bit groups. Each octal digit was translated from the 3-bit binary groups. Binary-to-Hexadecimal conversion is much the same:

```
BINARY TO HEXADECIMAL CONVERSION
Convert 10110111.1₂  to hexadecimal:
.
.                                      implied zeros
.                                           |||
.                               1011   0111   1000
Convert each group of bits      ----   ---- . ----
to its hexadecimal equivalent:    B      7      8
.
Answer:     10110111.1₂  =  B7.8₁₆
```

Here we had to group the bits in four's, from the binary point left, and from the binary point right, adding (implied) zeros as necessary to make complete 4-bit groups:

Likewise, the conversion from either octal or hexadecimal to binary is done by taking each octal or hexadecimal digit and converting it to its equivalent binary (3 or 4 bit) group, then putting all the binary bit groups together.

Incidentally, hexadecimal notation is more popular, because binary bit groupings in digital equipment are commonly multiples of eight (8, 16, 32, 64, and 128 bit), which are also multiples of 4.

Octal, being based on binary bit groups of 3, doesn't work out evenly with those common bit group sizings.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-1/octal-and-hexadecimal-numeration/

Although the prime intent of octal and hexadecimal numeration systems is for the "shorthand" representation of binary numbers in digital electronics, we sometimes need to convert from either of those systems to decimal form.

Of course, we could simply convert the hexadecimal or octal to binary, then convert from binary to decimal; since we know how to do both, but we can also convert directly.

Because octal is a base-eight numeration system, each place-weight value differs from either adjacent place by a factor of eight. For example, the octal number 245.37 can be broken down into place values as such:

```
octal
digits =       2  4  5  .  3  7
.              -  -  -  -  -  -
weight =       6  8  1     1  1
(in decimal    4           /  /
notation)                  8  6
.                             4
```

The decimal value of each octal place-weight times its respective cipher multiplier can be determined as follows:

(2 x 6410) + (4 x 810) + (5 x 110) + (3 x 0.12510) + (7 x 0.01562510) = 165.48437510

The technique for converting hexadecimal notation to decimal is the same, except that each successive place-weight changes *by a factor of sixteen*.

Simply denote each digit's weight, multiply each hexadecimal digit value by its respective weight (in decimal form), then add up all the decimal values to get a total. For example, the hexadecimal number 30F.A916 can be converted like this:

```
hexadecimal
digits =       3  0  F  .  A  9
.              -  -  -  -  -  -
weight =       2  1  1     1  1
(in decimal    5  6        /  /
notation)      6           1  2
.                          6  5
.                             6
```

$(3 \times 256_{10}) + (0 \times 16_{10}) + (15 \times 1_{10}) + (10 \times 0.0625_{10}) + (9 \times 0.00390625_{10}) = 783.66015625_{10}$

These basic techniques may be used to convert a numerical notation of any base into decimal form, if you know the value of that numeration system's base.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-1/octal-and-hexadecimal-to-decimal-conversion/

Because octal and hexadecimal has bases that are multiples of binary (base 2), conversion back and forth between either hexadecimal or octal and binary is very easy.

Also, because we are so familiar with the decimal system, converting binary, octal, or hexadecimal to decimal form is relatively easy (simply add up the products of cipher values and place-weights).

However, conversion from decimal to any of these "strange" numeration systems is a different matter.

The method which will probably make the most sense is the "trial-and-fit" method, where you try to "fit" the binary, octal, or hexadecimal notation to the desired value as represented in decimal form.

For example, let's say that I wanted to represent the decimal value of **87** in binary form. Let's start by drawing a binary number field, complete with place-weight values:

```
.
weight =       1  6  3  1  8  4  2  1
(in decimal    2  4  2  6
notation)      8
```

Well, we know that we won't have a "1" bit in the 128's place, because that would immediately give us a value greater than 87. However, since the next weight to the right (64) is less than 87, we know that we must have a "1" there.

```
.              1
.              -  -  -  -  -  -  -    Decimal value so far = 64₁₀
weight =       6  3  1  8  4  2  1
(in decimal    4  2  6
notation)
```

If we were to make the next place to the right a "1" as well, our total value would be 64_{10} + 32_{10}, or 96_{10}. This is greater than 87_{10}, so we know that this bit must be a "0". If we make the next (16's) place bit equal to "1," this brings our total value to 64_{10} + 16_{10}, or 80_{10}, which is closer to our desired value (87_{10}) without exceeding it:

```
.              1  0  1
.              -  -  -  -  -  -  -     Decimal value so far = 80₁₀
weight =       6  3  1  8  4  2  1
(in decimal    4  2  6
notation)
```

By continuing in this progression, setting each lesser-weight bit as we need to come up to our desired total value without exceeding it, we will eventually arrive at the correct figure:

```
.              1  0  1  0  1  1  1
.              -  -  -  -  -  -  -     Decimal value so far = 87₁₀
weight =       6  3  1  8  4  2  1
(in decimal    4  2  6
notation)
```

This trial-and-fit strategy will work with octal and hexadecimal conversions, too. Let's take the same 87_{10}, and convert it to **octal**:

```
.              -  -  -
weight =       6  8  1
(in decimal    4
notation)
```

If we put a cipher of "1" in the 64's place, we would have a total value of 64_{10} (less than 87_{10}). If we put a cipher of "2" in the 64's place, we would have a total value of 128_{10} (greater than 87_{10}). This tells us that our octal numeration must start with a "1" in the 64's place:

```
.              1
.              -  -  -     Decimal value so far = 64₁₀
weight =       6  8  1
(in decimal    4
notation)
```

Now, we need to experiment with cipher values in the 8's place to try and get a total (decimal) value as close to 87 as possible without exceeding it. Trying the first few cipher options, we get:

"1" = $64_{10} + 8_{10} = 72_{10}$
"2" = $64_{10} + 16_{10} = 80_{10}$
"3" = $64_{10} + 24_{10} = 88_{10}$

A cipher value of "3" in the 8's place would put us over the desired total of 87_{10}, so "2" it is!


```
.              1  2
.              -  -  -     Decimal value so far = 80₁₀
weight =       6  8  1
(in decimal    4
notation)
```

Now, all we need to make a total of 87 is a cipher of "7" in the 1's place:

```
.             1  2  7
.             -  -  -    Decimal value so far = 87₁₀
weight =      6  8  1
(in decimal   4
notation)
```

Of course, if you were paying attention during the last section on octal/binary conversions, you will realize that we can take the binary representation of 87_{10}, which we previously determined to be 1010111_2, and easily convert from that to octal to check our work:

```
.          Implied zeros
.               ||
.               001 010 111   Binary
.               _ ___ _
.                1   2   7    Octal
.
Answer: 1010111₂ = 127₈
```

Can we do decimal-to-hexadecimal conversion the same way? Sure, but who would want to? For *illustrative purposes.* this method is simple to understand, but laborious to carry out. There is better way to do these conversions; which is essentially the same (mathematically), but easier to accomplish.

This other method uses repeated cycles of division (using decimal notation) to break the decimal numeration down into multiples of binary, octal, or hexadecimal place-weight values.

In the first cycle of division, we take the original decimal number and divide it by the base of the numeration system that we're converting to (binary=2, octal=8, hex=16).

Then, we take the whole-number portion of division result (quotient) and divide it by the base value again, and so on; until we end up with a quotient of less than 1.

The binary, octal, or hexadecimal digits are determined by the "remainders" left over by each division step. Let's see how this works for binary, with the decimal example of 87_{10}:

```
 87              Divide 87 by 2, to get a quotient of 43.5
— = 43.5        Division "remainder" = 1, or the < 1 portion
 2                of the quotient times the divisor (0.5 x 2)

 43              Take the whole-number portion of 43.5 (43)
— = 21.5        and divide it by 2 to get 21.5, or 21 with
 2                a remainder of 1

 21              And so on . . . remainder = 1 (0.5 x 2)
— = 10.5
 2

 10              And so on . . . remainder = 0
— = 5.0
 2

 5               And so on . . . remainder = 1 (0.5 x 2)
— = 2.5
 2

 2               And so on . . . remainder = 0
— = 1.0
 2

 1               . . . until we get a quotient of less than 1
— = 0.5         remainder = 1 (0.5 x 2)
 2
```

The binary bits are assembled from the remainders of the successive division steps, beginning with the LSB and proceeding to the MSB. In this case, we arrive at a binary notation of 1010111_2. When we divide by 2, we will always end up with either ".0" or ".5" (i.e. a remainder of either 0 or 1).

As was said before, this repeat-division technique for conversion will work for numeration systems other than binary. If we were to perform successive divisions using a different number, such as 8 for conversion to octal, we will necessarily get remainders between 0 and 7. Let's try this with the same decimal number, 87_{10}:

```
.   87              Divide 87 by 8, to get a quotient of 10.875
.  — = 10.875      Division "remainder" = 7, or the < 1 portion
.   8                of the quotient times the divisor (.875 x 8)
.
.   10
.  — = 1.25        Remainder = 2
.   8
.
.   1
.  — = 0.125       Quotient is less than 1, so we'll stop here.
.   8                Remainder = 1
.
.  RESULT:   87₁₀   = 127₈
```

We can use a similar technique for converting numeration systems dealing with quantities less than 1, as well. For converting a decimal number less than 1 into binary, octal, or

hexadecimal, we use *repeated multiplication* ; taking the integer portion of the product in each step as the next digit of our converted number. Let's use the decimal number 0.8125_{10} as an example, converting to binary:

```
0.8125 x 2 = 1.625      Integer portion of product = 1

0.625 x 2 = 1.25        Take < 1 portion of product and remultiply
                        Integer portion of product = 1

0.25 x 2 = 0.5          Integer portion of product = 0

0.5 x 2 = 1.0           Integer portion of product = 1
                        Stop when product is a pure integer
                        (ends with .0)

RESULT:   0.8125₁₀   = 0.1101₂
```

As with the repeat-division process for integers, each step gives us the next digit (or bit) further away from the "point." With integer (division), we worked from the LSB to the MSB (right-to-left).

But with repeated multiplication, we worked from the left to the right. To convert a decimal number > 1, with a < 1 component, we must use both techniques, one at a time. Take the decimal example of 54.40625_{10}, converting to binary:

```
REPEATED DIVISION FOR THE INTEGER PORTION:
.
.     54
.    — = 27.0          Remainder = 0
.     2
.
.     27
.    — = 13.5          Remainder = 1  (0.5 x 2)
.     2
.
.     13
.    — = 6.5           Remainder = 1 (0.5 x 2)
.     2
.
.     6
.    — = 3.0           Remainder = 0
.     2
.
.     3
.    — = 1.5           Remainder = 1 (0.5 x 2)
.     2
.
.     1
.    — = 0.5           Remainder = 1 (0.5 x 2)
.     2
.
PARTIAL ANSWER:  54₁₀   = 110110₂
```

```
REPEATED MULTIPLICATION FOR THE < 1 PORTION:
.
.  0.40625 x 2 = 0.8125  Integer portion of product = 0
.
.  0.8125 x 2 = 1.625    Integer portion of product = 1
.
.  0.625 x 2 = 1.25      Integer portion of product = 1
.
.  0.25 x 2 = 0.5        Integer portion of product = 0
.
.  0.5 x 2 = 1.0         Integer portion of product = 1
.
.  PARTIAL ANSWER:  0.40625₁₀  = 0.01101₂
.
.  COMPLETE ANSWER: 54₁₀  + 0.40625₁₀  = 54.40625₁₀
.
.                   110110₂ + 0.01101₂ = 110110.01101₂
```

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-1/conversion-from-decimal-numeration/

Binary Arithmetic

Binary Addition

Adding binary numbers is a very simple task, and very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10
1 + 1 + 1 = 11

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is “carried” to the next left column. Consider the following examples:

```
                    11  1 <--- Carry bits -----> 11
  1001101            1001001                    1000111
+ 0010010          + 0011001                  + 0010110
---------          ---------                  ---------
  1011111            1100010                    1011101
```

The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-2/binary-addition/

Negative Binary Numbers

With addition being easily accomplished, we can perform the operation of subtraction with the same technique simply by making one of the numbers negative.

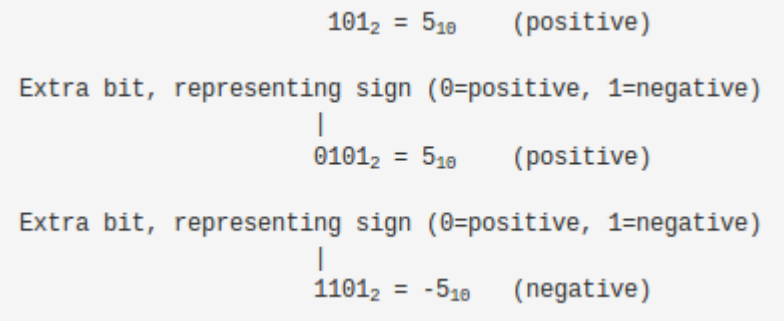
For example, the subtraction problem of 7 - 5 is essentially the same as the addition problem 7 + (-5).

Since we already know how to represent positive numbers in binary, all we need to know now is how to represent their negative counterparts and we'll be able to subtract.

Usually we represent a negative decimal number by placing a minus sign directly to the left of the most significant digit, just as in the example above, with -5.

However, the whole purpose of using binary notation is for constructing on/off circuits that can represent bit values in terms of voltage (2 alternative values: either "high" or "low").

In this context, we don't have the luxury of a third symbol such as a "minus" sign, since these circuits can only be on or off (two possible states). One solution is to reserve a bit (circuit) that does nothing but represent the mathematical sign:



As you can see, we have to be careful when we start using bits for any purpose other than standard place-weighted values. Otherwise, 1101_2 could be misinterpreted as the number thirteen when in fact we mean to represent negative five.

To keep things straight here, we must first decide how many bits are going to be needed to represent the largest numbers we'll be dealing with, and then be sure not to exceed that bit field length in our arithmetic operations.

For the above example, I've limited myself to the representation of numbers from negative seven (1111_2) to positive seven (0111_2), and no more, by making the fourth bit the "sign" bit.

Only by first establishing these limits can I avoid confusion of a negative number with a larger, positive number.

Representing negative five as 1101_2 is an example of the sign-magnitude system of negative binary numeration.

By using the leftmost bit as a sign indicator and not a place-weighted value, I am sacrificing the "pure" form of binary notation for something that gives me a practical advantage: the representation of negative numbers. The leftmost bit is read as the sign, either positive or negative, and the remaining bits are interpreted according to the standard binary notation (left to right, in multiples of two).

As simple as the sign-magnitude approach is, it is not very practical for arithmetic purposes.

For instance, how do I add a negative five (1101_2) to any other number, using the standard technique for binary addition?

I'd have to invent a new way of doing addition in order for it to work, and if I do that, I might as well just do the job with longhand subtraction; there's no arithmetical advantage to using negative numbers to perform subtraction through addition if we have to do it with sign-magnitude numeration, and that was our goal!

There's another method for representing negative numbers which works with our familiar technique of longhand addition; and also happens to make more sense from a place-weighted numeration point of view, called **complementation**.

With this strategy, we assign *the leftmost bit to serve a special purpose*, just as we did with the sign-magnitude approach; defining our number limits just as before.

However, this time, the leftmost bit is more than just a sign bit; rather, it possesses a negative place-weight value.

For example, a value of negative five would be represented as such:

```
Extra bit, place weight = negative eight
.                    |
.                    1011₂ = 5₁₀   (negative)
.
.          (1 x -8₁₀)  +  (0 x 4₁₀)  +  (1 x 2₁₀)  +  (1 x 1₁₀)  =  -5₁₀
```

With the right three bits being able to represent a magnitude from zero through seven, and the leftmost bit representing either zero or negative eight, we can successfully represent any integer number from negative seven ($1001_2 = -8_{10} + 1_{10} = -7_{10}$) to positive seven ($0111_2 = 0_{10} + 7_{10} = 7_{10}$).

Representing positive numbers in this scheme (with the fourth bit designated as the negative weight) is no different from that of ordinary binary notation.

However, representing negative numbers is not quite as straightforward:

```
zero            0000
positive one    0001          negative one     1111
positive two    0010          negative two     1110
positive three  0011          negative three   1101
positive four   0100          negative four    1100
positive five   0101          negative five    1011
positive six    0110          negative six     1010
positive seven  0111          negative seven   1001
.                             negative eight   1000
```

Note that the negative binary numbers in the right column, being the sum of the right three bits' total plus the negative eight of the leftmost bit, don't "count" in the same progression as the positive binary numbers in the left column.

Rather, the right three bits have to be set at the proper value to equal the desired (negative) total when summed with the negative eight place value of the leftmost bit.

Those right three bits are referred to as the **two's complement** of the corresponding positive number. Consider the following comparison:

```
positive number       two's complement
---------------       ----------------
001                    111
010                    110
011                    101
100                    100
101                    011
110                    010
111                    001
```

In this case, with the negative weight bit being the fourth bit (place value of negative eight), the two's complement *for any positive number* will be whatever value is needed to add to negative eight to make that positive value's negative equivalent.

Thankfully, there's an easy way to figure out the two's complement for any binary number: simply invert all the bits of that number, changing all 1's to 0's and vice versa (to arrive at what is called the one's complement) and then add one!

For example, to obtain the two's complement of five (101_2), we would first invert all the bits to obtain 010_2 (the "one's complement"), then add one to obtain 011_2, or -5_{10} in three-bit/two's complement form.

Interestingly enough, generating the two's complement of a binary number works the same if you manipulate all the bits, including the leftmost (sign) bit at the same time as the magnitude bits.

Let's try this with the former example, converting a positive five to a negative five, but performing the complementation process on all four bits.

We must be sure to include the 0 (positive) sign bit on the original number, five (0101_2). First, inverting all bits to obtain the one's complement: 1010_2. Then, adding one, we obtain the final answer: 1011_2, or -5_{10} expressed in four-bit, two's complement form.

It is critically important to remember that the place of the negative-weight bit **must be already determined** *before* any two's complement conversions can be done.

If our binary numeration field were such that the eighth bit was designated as the negative-weight bit (10000000_2), we'd have to determine the two's complement based on *all seven of the other bits*. Here, the two's complement of five (0000101_2) would be 1111011_2. A positive five in this system would be represented as 00000101_2, and a negative five as 11111011_2.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-2/negative-binary-numbers/

Binary Subtraction

We can subtract one binary number from another by using the standard techniques adapted for decimal numbers (subtraction of each bit pair, right to left, "borrowing" as needed from bits to the left).

However, if we can leverage the already familiar (and easier) technique of binary addition to subtract, that would be better. As we just learned, we can represent negative binary numbers by using the "two's complement" method (and a negative place-weight bit).

Here, we'll use those negative binary numbers to subtract through addition. Here's a sample problem: Subtraction: $7_{10} - 5_{10}$ Addition equivalent: $7_{10} + (-5_{10})$

If all we need to do is represent seven and negative five in binary (two's complemented) form, all we need is three bits plus the negative-weight bit: positive seven = 0111_2; negative five = 1011_2

Now, let's add them together:

```
        1111  <--- Carry bits
         0111
       + 1011
       ------
        10010
        |
 Discard extra bit

Answer = 0010₂
```

Since we've already defined our number bit field as three bits plus the negative-weight bit, the fifth bit in the answer (1) will be discarded to give us a result of 0010_2, or positive two: the correct answer.

Another way to understand why we discard that extra bit is to *remember that the leftmost bit of the lower number possesses a negative weight*, in this case equal to negative eight.

When we add these two binary numbers together, what we're actually doing with the MSBs is subtracting the lower number's MSB from the upper number's MSB. In subtraction, one never "carries" a digit or bit on to the next left place-weight. Let's try another example, this time with larger numbers.

If we want to add -25_{10} to 18_{10}, we must first decide how large our binary bit field must be. To represent the largest (absolute value) number in our problem, which is twenty-five, we need at least five bits, plus a sixth bit for the negative-weight bit.

Let's start by representing positive twenty-five, then finding the two's complement and putting it all together into one numeration:

$+25_{10} = 011001_2$ (showing all six bits)
One's complement of $11001_2 = 100110_2$
One's complement + 1 = two's complement = 100111_2
$-25_{10} = 100111_2$

Essentially, we're representing negative twenty-five by using the negative-weight (sixth) bit with a value of negative thirty-two, plus positive seven (binary 111_2). Now, let's represent positive eighteen in binary form, showing all six bits:

$18_{10} = 010010_2$

Now, let's add them together and see what we get:

```
    11   <--- Carry bits
  100111
+ 010010
--------
  111001
```

Since there were no "extra" bits on the left, there are no bits to discard. The leftmost bit on the answer is a 1, which means that the answer is negative, in two's complement form, as it should be. Converting the answer to decimal form by summing all the bits times their respective weight values, we get: $(1 \times -32_{10}) + (1 \times 16_{10}) + (1 \times 8_{10}) + (1 \times 1_{10}) = -7_{10}$

Indeed -7^{10} is the proper sum of -25_{10} and 18_{10}.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-2/binary-subtraction/

Binary Overflow

One caveat with signed binary numbers is that of **overflow**, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the alloted number of bits.

Remember that the place of the sign bit is fixed from the beginning of the problem. With the last example problem, we used five binary bits to represent the magnitude of the number, and the left-most (sixth) bit as the negative-weight, or sign, bit.

With five bits to represent magnitude, we have a representation range of 25, or thirty-two integer steps from 0 to maximum. This means that we can represent a number as high as $+31_{10}$ (011111_2), or as low as -32_{10} (100000_2).

If we set up an addition problem with two binary numbers, the sixth bit used for sign, and the result either exceeds $+31_{10}$ or is less than -32_{10}, our answer will be incorrect.

Let's try adding 17_{10} and 19_{10} to see how this overflow condition works for excessive positive numbers:

$17_{10} = 10001_2$ $\qquad$ $19_{10} = 10011_2$

```
                      1  11  <--- Carry bits
(Showing sign bits)   010001
                    + 010011
                    --------
                      100100
```

The answer (100100_2), interpreted with the sixth bit as the -32_{10} place, is actually equal to -28_{10}, not $+36_{10}$ as we should get with $+17_{10}$ and $+19_{10}$ added together.

Obviously, this is not correct. What went wrong? The answer lies in the restrictions of the six-bit number field within which we're working. Since the magnitude of the true and proper sum (36_{10}) exceeds the allowable limit for our designated bit field, we have an overflow error.

Simply put, six places doesn't give enough bits to represent the correct sum, so whatever figure we obtain using the strategy of discarding the left-most "carry" bit will be incorrect.

A similar error will occur if we add two negative numbers together to produce a sum that is too low for our six-bit binary field.

Let's try adding -17_{10} and -19_{10} together to see how this works (or doesn't work, as the case may be)...

```
.      -17₁₀  = 101111₂          -19₁₀  = 101101₂
.
.                            1 1111  <--- Carry bits
.    (Showing sign bits)      101111
.                           + 101101
.                           --------
.                            1011100
.                            |
.                   Discard extra bit
.
FINAL ANSWER:   011100₂   = +28₁₀
```

The (incorrect) answer is a positive twenty-eight. The sixth sign bit is the root cause of this difficulty.

Let's try these two problems again, except this time using the seventh bit for a sign bit, and allowing the use of 6 bits for representing the magnitude:

```
       17₁₀ + 19₁₀                        (-17₁₀) + (-19₁₀)

        1  11                               11 1111
       0010001                              1101111
     + 0010011                            + 1101101
     ---------                            ---------
       0100100₂                            11011100₂
                                           |
                                  Discard extra bit

ANSWERS:  0100100₂ = +36₁₀
          1011100₂ = -36₁₀
```

By using bit fields sufficiently large to handle the magnitude of the sums, we arrive at the correct answers. In these sample problems we've been able to detect overflow errors by performing the addition problems in decimal form and comparing the results with the binary answers.

For example, when adding $+17_{10}$ and $+19_{10}$ together, we knew that the answer was supposed to be $+36_{10}$, so when the binary sum checked out to be -28_{10}, we knew that something had to be wrong.

Although this is a valid way of detecting overflow, it is not very efficient. After all, the whole idea of complementation is to be able to *reliably add binary numbers together* and not have to double-check the result by adding the same numbers together in decimal form!

This is especially true for the purpose of building electronic circuits to add binary quantities together; the circuit has to be able to check itself for overflow without the supervision of a

human being who already knows what the correct answer is. What we need is a simple error-detection method that doesn't require any additional arithmetic.

Perhaps the most elegant solution is to check for the sign of the sum and compare it against the signs of the numbers added. Obviously, two positive numbers added together should give a positive result, and two negative numbers added together should give a negative result.

Notice that whenever we had a condition of overflow in the example problems, the sign of the sum was always opposite of the two added numbers: $+17_{10}$ plus $+19_{10}$ giving -28_{10}, or -17_{10} plus -19_{10} giving $+28_{10}$.

By checking the signs alone we are able to tell that something is wrong. But what about cases where a positive number is added to a negative number? What sign should the sum be in order to be correct?

Or, more precisely, what sign of sum would *necessarily indicate* an overflow error?

The answer to this is equally elegant: there will never be an overflow error when two numbers of opposite signs are added together! The reason for this is apparent when the nature of overflow is considered.

Overflow occurs when the magnitude of a number exceeds *the range allowed by the size of the bit field*.

The sum of two identically-signed numbers may very well exceed the range of the bit field of those two numbers; and so in this case, overflow is a possibility. However, if a positive number is added to a negative number, the sum will always be closer to zero than either of the two added numbers.

Hence its magnitude must be *less than the magnitude of either original number*, and so overflow is impossible. Fortunately, this technique of overflow detection is easily implemented in electronic circuitry, and it is a standard feature in **digital adder circuits**

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-2/binary-overflow/

Digital Signals & Logic Gates

What makes binary numeration so important to the application of digital electronics is the ease in which bits may be represented in physical terms. Because a binary bit can only have one of two different values (0 or 1), any physical medium capable of switching between two saturated states may be used to represent a bit.

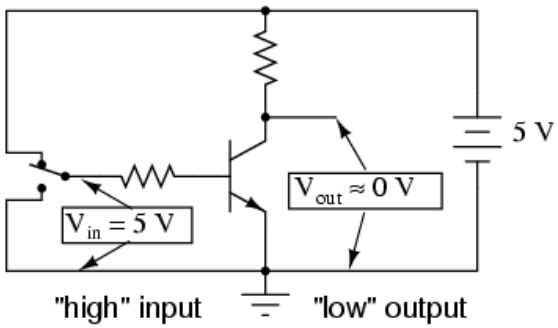
Consequently, any physical system capable of representing binary bits is able to represent numerical quantities and potentially has the ability to manipulate those numbers. This is the basic concept underlying digital computing.

Transistors, when operated at their bias limits, may be in one of two different states: either cut off (no controlled current) or saturation (maximum controlled current).

If a transistor circuit is designed to maximize the probability of falling into either one of these states (and not operating in the linear, or active, mode), it can serve as a physical representation of a binary bit.

A voltage signal measured at the output of such a circuit may also serve as a representation of a single bit, a low voltage representing a binary “0” and a (relatively) high voltage representing a binary “1”. Note the following transistor circuit:

Transistor in saturation



0 V = "low" logic level (0)

5 V = "high" logic level (1)

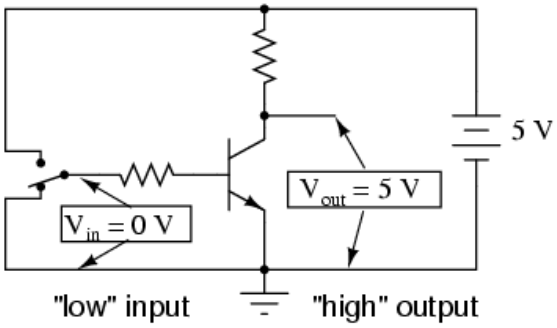
In this circuit, the transistor is in a state of saturation by virtue of the applied input voltage (5 volts) through the two-position switch. Because its saturated, the transistor drops very little voltage between collector and emitter, resulting in an output voltage of (practically) 0 volts.

If we were using this circuit to represent binary bits, we would say that the input signal is a binary “1” and that the output signal is a binary “0.” Any voltage close to full supply voltage (measured in reference to ground, of course) is considered a “1” and a lack of voltage is considered a “0.”

Alternative terms for these voltage levels are high (same as a binary “1”) and low (same as a binary “0”). A general term for the representation of a binary bit by a circuit voltage is **logic level**.

Moving the switch to the other position, we apply a binary “0” to the input and receive a binary “1” at the output:

Transistor in cutoff

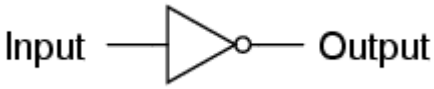


0 V = "low" logic level (0)
5 V = "high" logic level (1)

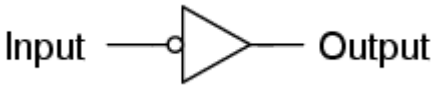
This single-transistor logic gate is a special type of amplifier circuit designed to accept and generate voltage signals corresponding to binary 1's and 0's. As such, gates are not intended to be used for amplifying analog signals (voltage signals between 0 and full voltage).

The gate shown here with the single transistor is known as an **inverter** (NOT gate) because it outputs the exact opposite digital signal as what is input. For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors. The following is the symbol for an inverter:

Inverter, or NOT gate



An alternative symbol for an inverter is shown here:



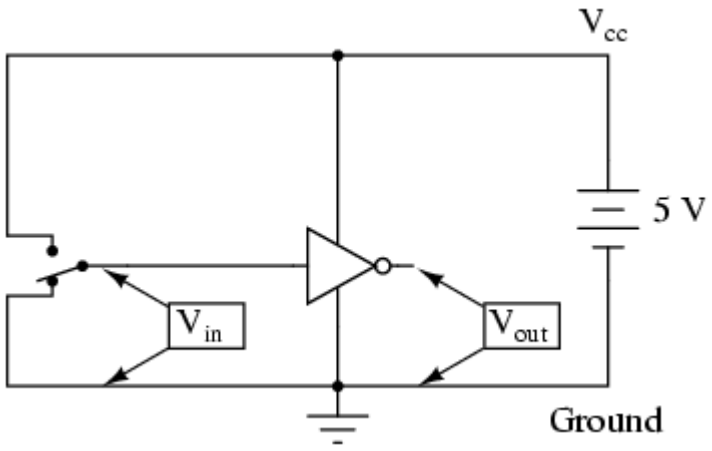
Notice the triangular shape of the gate symbol, much like that of an **operational amplifier** ("op amp").

The small circle or "bubble" shown on either the input or output terminal is standard for representing the inversion function. Remove the bubble from the gate symbol (leaving only a triangle), and the resulting symbol would no longer indicate inversion, but rather *direct amplification* (aka **buffer**).

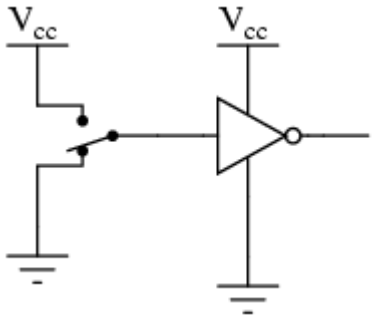
Like an operational amplifier symbol, input and output connections are shown as single wires, the implied reference point for each voltage signal being "ground." In digital gate

circuits, ground is almost always the negative connection of a single voltage source (power supply).

Like operational amplifiers, the power supply connections for digital gates are often omitted from the symbol for simplicity's sake. If we were to show all the necessary connections needed for operating this gate, the schematic would look something like this:



Power supply conductors are rarely shown in gate circuit schematics, even if the power supply connections at each gate are. Minimizing lines in our schematic, we get this:



"**V_{cc}**" stands for the constant voltage *supplied to the collector* of a bipolar junction transistor circuit, in reference to ground. Those points in a gate circuit marked by the label "V_{cc}" are all connected to the same point, and that point is the positive terminal of a DC voltage source, usually 3.3 or 5 volts.

There are actually quite a few different types of logic gates, most of which have multiple input terminals for accepting more than one signal. The output of any gate is dependent on the state of its input(s) and its logical function.

One common way to express the particular function of a gate circuit is called a **truth table**. Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low"/ "1" or "0," for each gate input terminal), along with the corresponding output logic level; either "high" or "low."

For the previous inverter (NOT gate) circuit illustrated, the truth table looks like this:

NOT gate truth table

Input	Output
0	1
1	0

Truth tables for more complex gates are, of course, larger than the one shown for the NOT gate. A gate's truth table must have as many rows as there are possibilities for unique input combinations. A single-input gate like the NOT gate, has only two possibilities, 0 and 1.

However, for a two input gate, there are four possibilities (00, 01, 10, and 11), and thus four rows to the corresponding truth table. For a three-input gate, there are eight possibilities (000, 001, 010, 011, 100, 101, 110, and 111), and thus an eight-row truth table.

Bonus points if you realized that the number of truth table rows needed for a gate is equal to 2 raised to the power of the number of input terminals.

Brief Recap:

- In digital circuits, binary bit values of 0 and 1 are represented by voltage signals measured in reference to a common circuit point called ground; an absence of voltage represents a binary "0" and the presence of full DC supply voltage represents a binary "1".
- A logic gate is a special form of amplifier circuit designed to input and output logic level voltages (voltages intended to represent binary bits).
- Gate circuits are most commonly represented in a schematic by their own unique symbols rather than by their constituent transistors and resistors.
- Just as with operational amplifiers, the power supply connections to gates are often omitted in schematic diagrams for the sake of simplicity.
- A truth table is a standard way of representing the input/output relationships of a gate circuit, listing all the possible input logic level combinations with their respective output logic levels.

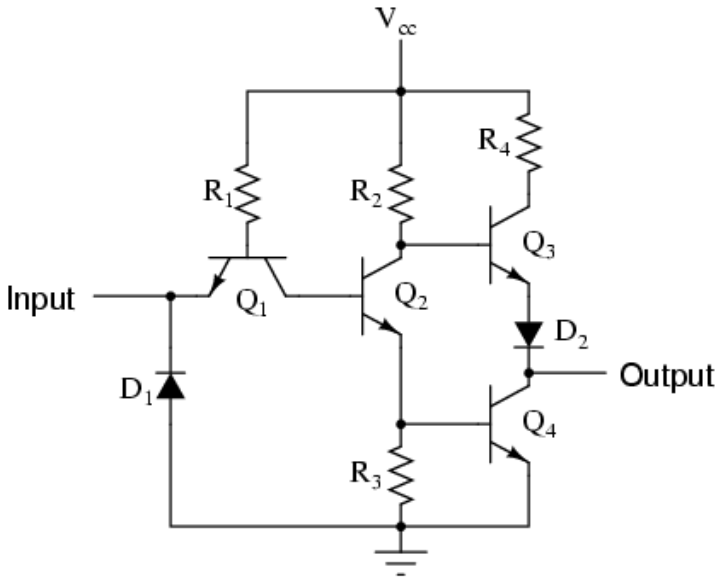
Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-3/digital-signals-gates/

Practical Gates

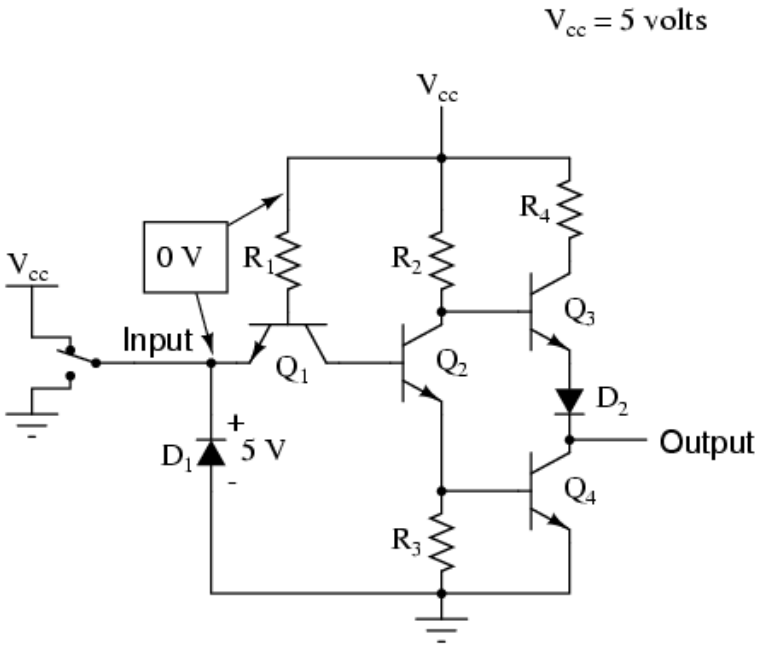
The single-transistor inverter circuit illustrated earlier is far too crude to be of practical use.

Real inverter circuits contain more than one transistor to maximize voltage gain (ensuring final output is either in full cutoff or full saturation), plus circuitry for mitigating accidental damage. Here's a schematic for a efficient and reliable real-world inverter circuit.
Let's analyze this circuit for the condition where the input is "high," or in a binary "1" state.

Practical inverter (NOT) circuit



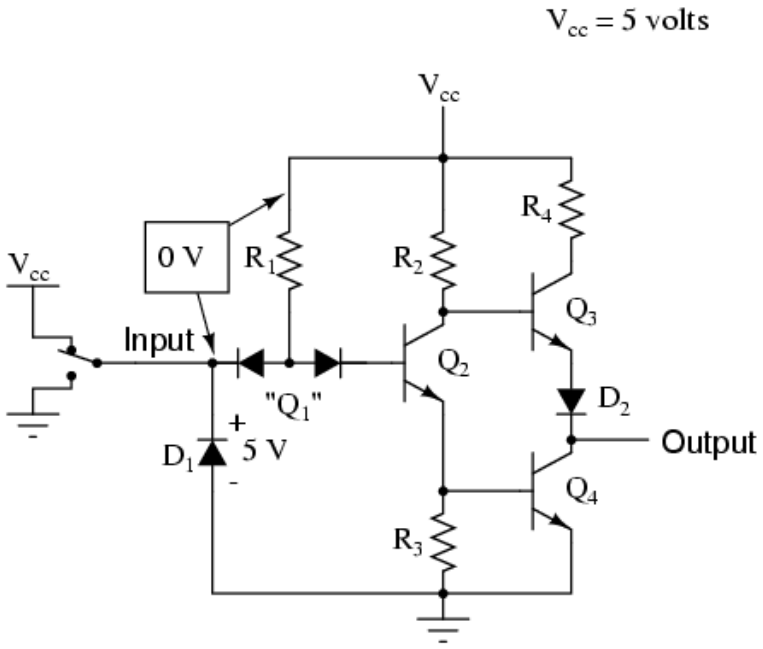
We can simulate this by showing the input terminal connected to V_{cc} through a switch:



In this case, diode D1 will be *reverse-biased*, and therefore not conduct any current. In fact, the only purpose for having D1 in the circuit is to prevent transistor damage in the case of a negative voltage being impressed on the input (a voltage that is negative, rather than positive, with respect to ground).

Note that transistor Q1 is not being used in a typical transistor configuration. In reality, Q1 is being used in this circuit as nothing more than a back-to-back **pair of diodes**.

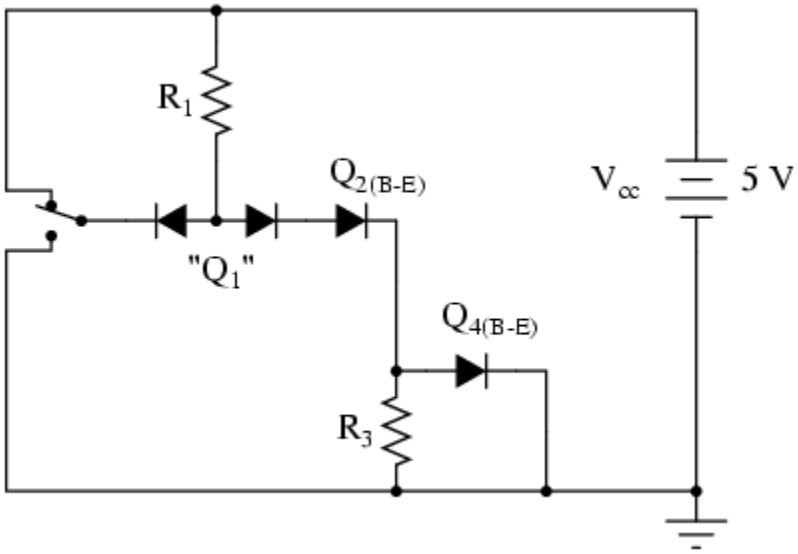
The following schematic shows the real function of Q1:



While the purpose of these diodes is to "steer" current to or away from the base of transistor Q2, depending on the logic level of the input… **how** these two diodes are able to "steer" current isn't exactly obvious at first glance.

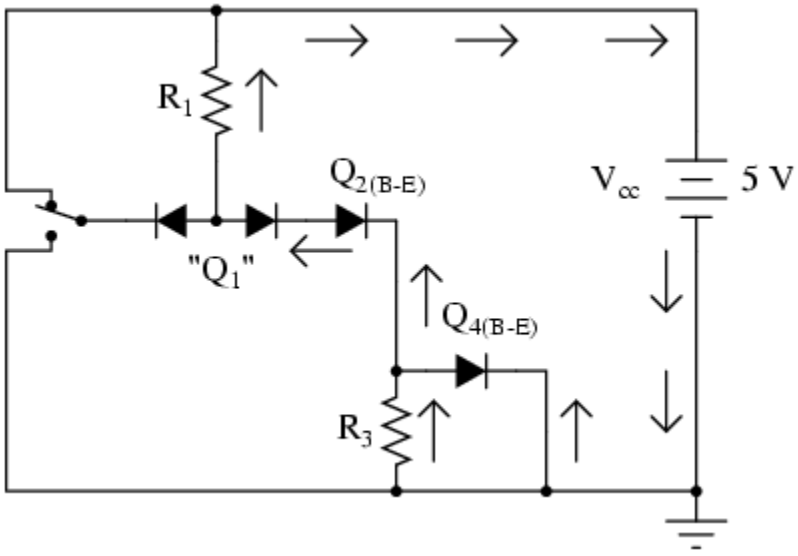
Hence a short example is in order.

Let's suppose we had the following diode/resistor circuit, representing the base-emitter junctions of transistors Q2 and Q4 as *single diodes*; with all other portions of the circuit stripped away so that we can concentrate on the current "steered" through the two back-to-back diodes:



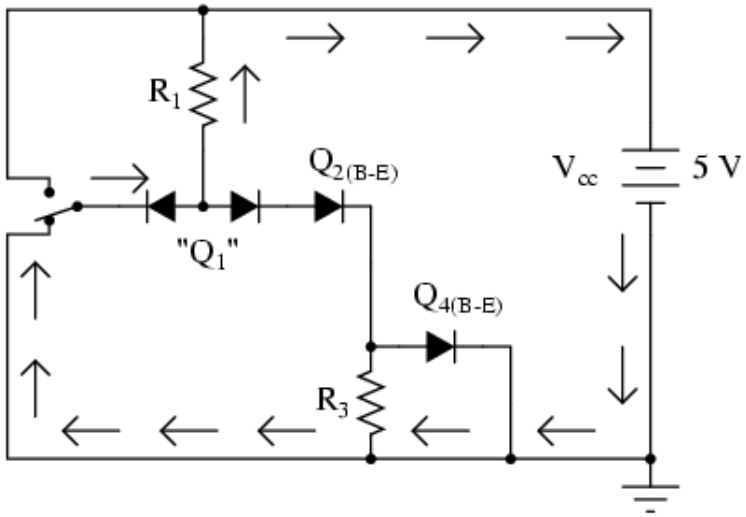
With the input switch in the "up" position (connected to V_{cc}), it should be obvious that there will be no current through the left steering diode of Q1; because there isn't any voltage in the switch-diode-R1-switch loop to motivate electrons to flow.

However, there will be current *through the right steering diode* of Q1, as well as through Q2's base-emitter diode junction and Q4's base-emitter diode junction:



This tells us that in the real gate circuit, transistors Q2 and Q4 will have **base current**, which will turn them on to *conduct collector current*. The total voltage dropped between the base of Q1 (the node joining the two back-to-back steering diodes) and ground will be ~2.1 volts, via the combined voltage drops of three PN junctions (right steering diode, Q2 base-emitter diode, and Q4 base-emitter diode).

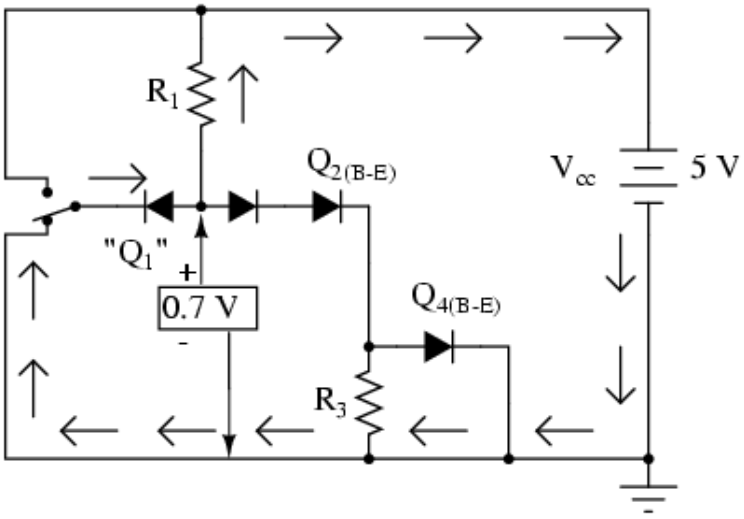
Now, let's move the input switch to the "down" position and see what happens:



If we were to *measure current* in this circuit, it appears as though there is a complete path for current through diode's Q4 and Q2, the right diode of the pair, and R1... so why no current through that path?!

Remember that PN junction diodes are very nonlinear devices: they do not even begin to conduct current until the forward voltage applied across them reaches a certain threshold (~0.7 volts for silicon and ~0.3 volts for germanium).

And then when they begin to conduct current, they will not drop substantially more than 0.7 volts. When the switch in this circuit is in the "down" position, the left diode of the steering diode pair is fully conducting, and so it drops about 0.7 volts across it and no more.



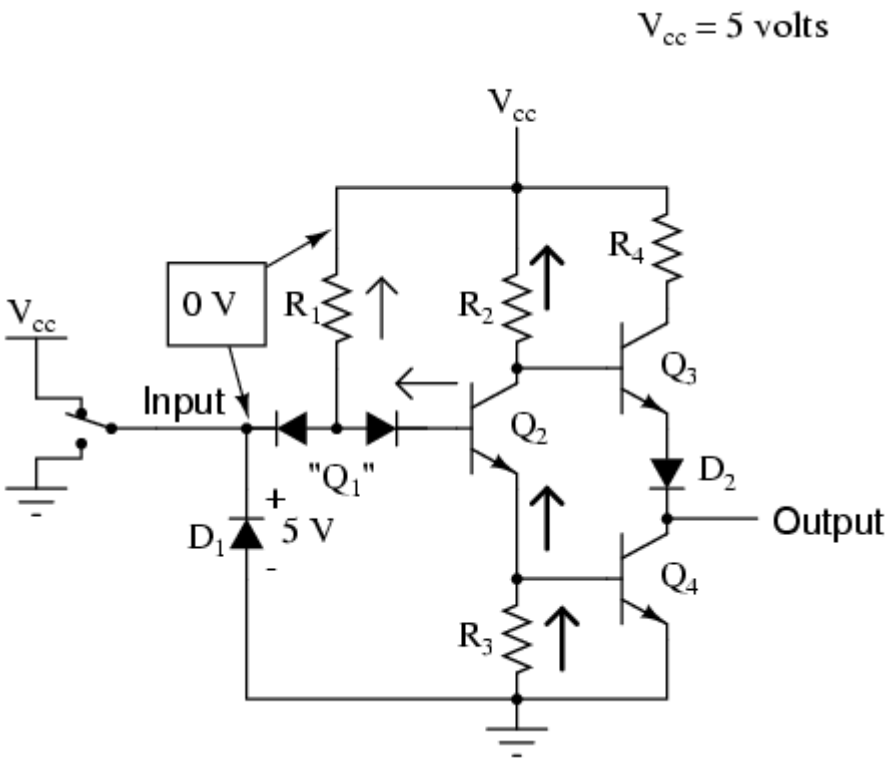
Recall that with the switch in the "up" position (i.e. Q2 and Q4 conducting), there was ~2.1 volts dropped between those same two points (Q1's base and ground), which also just happens to be the minimum voltage necessary to forward-bias three series-connected silicon PN junctions into a state of conduction.

The 0.7 volts provided by the left diode's forward voltage drop is simply insufficient to allow any electron flow through the series string of the right diode, Q2's diode, and the R3//Q4 diode parallel sub-circuit; hence no electrons flow through that path.

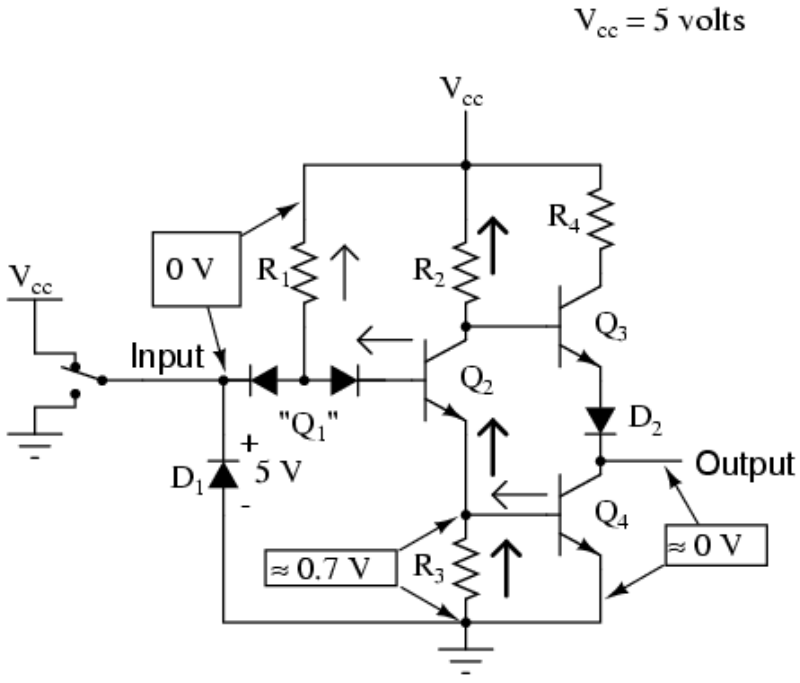
With no current through the bases of either transistor Q2 or Q4, neither one will be able to conduct collector current: transistors Q2 and Q4 will both be in a state of **cutoff**.

Consequently, this circuit configuration allows 100 percent switching of Q2 base current (and therefore control over the rest of the gate circuit, including voltage at the output) by diversion of current through the left steering diode.

In the case of our example gate circuit, the input is held "high" by the switch (connected to V_{cc}), making the left steering diode (zero voltage dropped across it). However, the right steering diode is conducting current through the base of Q2, through resistor R1:

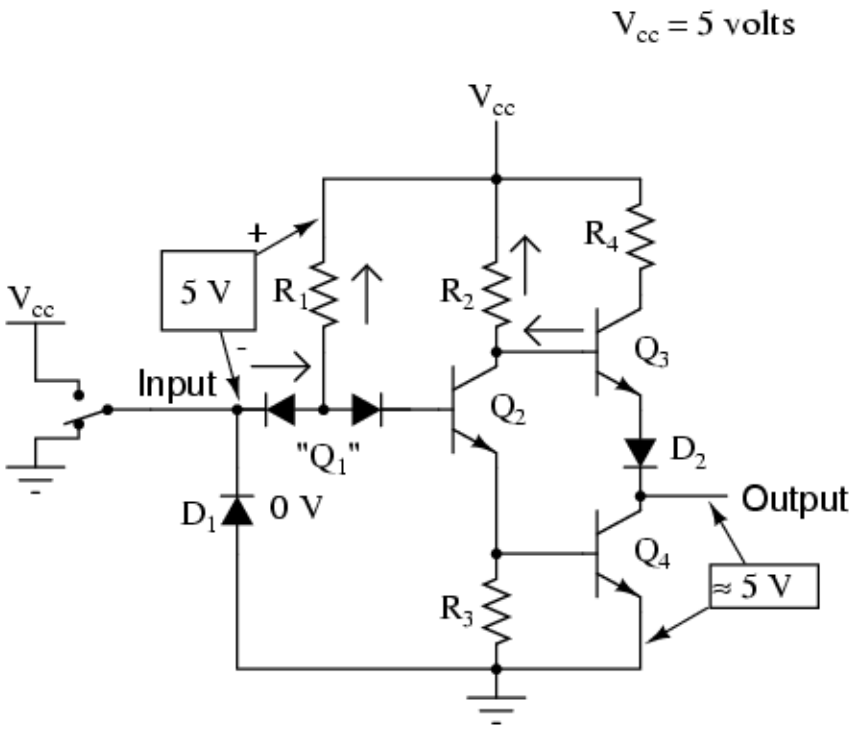


With base current provided, transistor Q2 will be turned “on.” More specifically, it will be saturated by virtue of the more-than-adequate current allowed by R1 through the base. With Q2 saturated, resistor R3 will be dropping enough voltage to forward-bias the base-emitter junction of transistor Q4, thus saturating it as well:



With Q4 saturated, the output terminal will be almost directly shorted to ground; thus leaving the output terminal at a voltage (relative to ground) of almost 0 volts, or a binary "0" ("low") logic level.

Due to the presence of diode D2, there will not be enough voltage between the base of Q3 and its emitter to turn it on, so it remains in cutoff. So now let's see what happens if we *reverse the input's logic level* to a binary "0" by actuating the input switch:

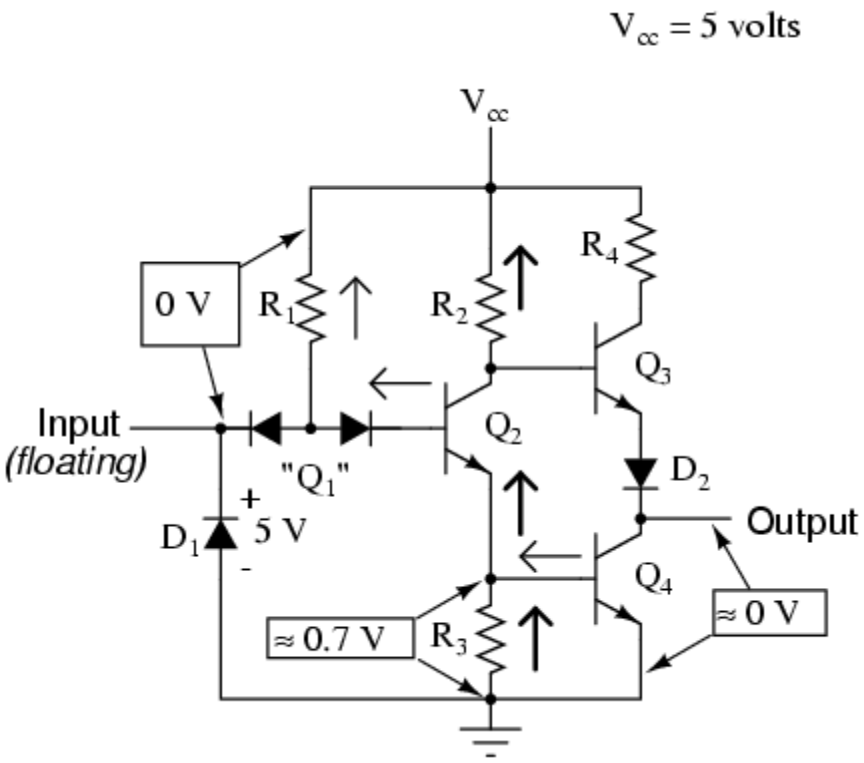


Now there will be current through the left steering diode of Q1 and no current through the *right steering diode*. This eliminates current through the base of Q2, thus turning it off. With Q2 off, there is no longer a path for Q4 base current, so Q4 goes into cutoff as well.

Q3, on the other hand, now has sufficient voltage dropped between its base and ground to *forward-bias its base-emitter junction and saturate it*, thus raising the output terminal voltage to a "high" state. In actuality, the output voltage will be somewhere around 4 volts depending on the degree of saturation (and any load current)... but still high enough to be considered a "high" (1) logic level.

With this, we have a *realistic simulation* of the inverter circuit: a "1" in gives a "0" out, and vice versa.

The astute observer will note that this inverter circuit's input will assume a "high" state of left floating (not connected to either V_{cc} or ground). With the input terminal left unconnected, there will be no current through the left steering diode of Q1, leaving all of R1's current to go through Q2's base; thusly saturating Q2 and driving the circuit output to a "low" state:

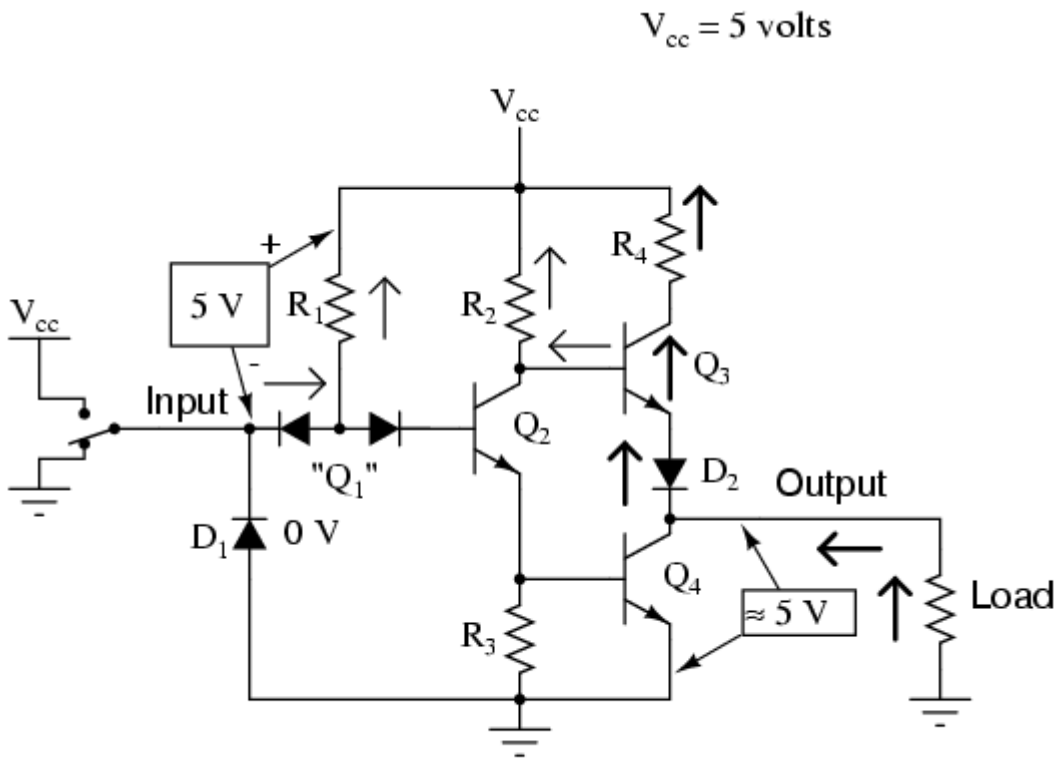


The tendency for such a circuit to assume a high input state (if left floating) is one shared by all gate circuits based on this type of design, known as Transistor-to-Transistor Logic (**TTL**).

This characteristic may be taken advantage of in simplifying the design of a gate's output circuitry; knowing that the outputs of gates *typically drive the inputs of other gates*. If the input of a TTL gate circuit assumes a high state when floating, then the output of any gate driving a TTL input need only provide a path to ground for a low state and be floating for a high state.

Thus this type of gate circuit has the ability to *handle output current in two directions*: in and out.

This is known as **sourcing** and **sinking** current, respectively. When the gate output is high, there is continuity from the output terminal to V_{cc} through the top output transistor (Q3), allowing electrons to flow from ground, through a load, into the gate's output terminal, through the emitter of Q3, and eventually up to the V_{cc} power terminal (positive side of the DC power supply):

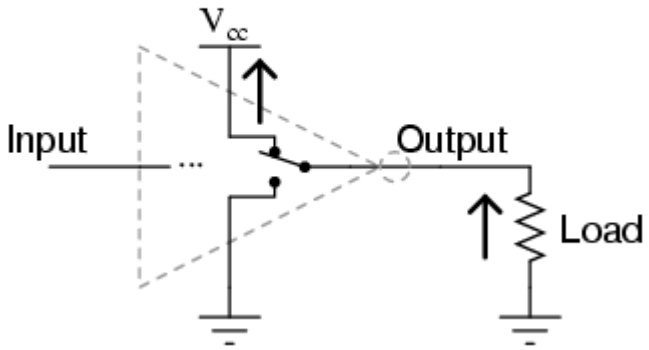


*Inverter gate **sourcing** current*

To simplify this concept, we may show the output of a gate circuit as being a double-throw switch, capable of connecting the output terminal either to V_{cc} or ground, depending on its state.

For a gate outputting a "high" logic level, the combination of Q3 saturated and Q4 cutoff is analogous to a double-throw switch in the "V_{cc}" position, providing a path for current through a grounded load:

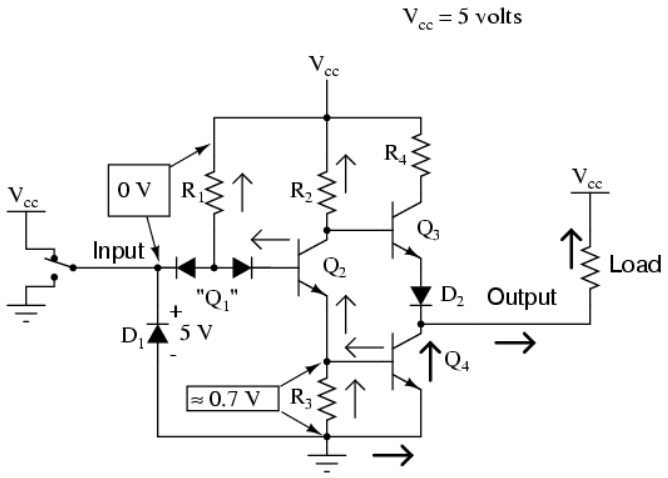
*Simplified gate circuit **sourcing** current*



Please note that this two-position switch shown inside the gate symbol is representative of transistors Q3 and Q4 alternately connecting the output terminal to V_{cc} or ground, not of the switch previously shown sending an input signal to the gate!

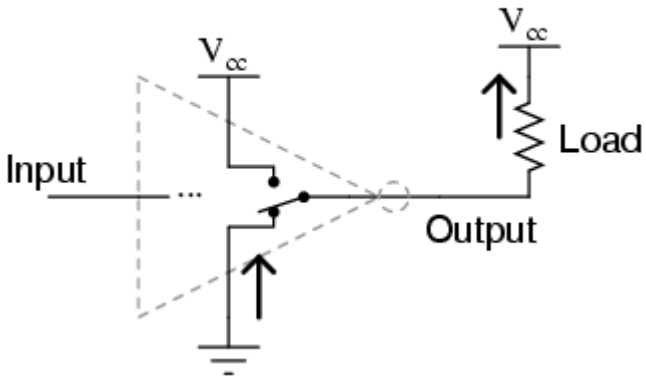
Conversely, when a gate circuit is outputting a “low” logic level to a load, it is analogous to the double-throw switch being set in the “ground” position.

Current will then be going the other way if the load resistance connects to V_{cc}: from ground, through the emitter of Q4, out the output terminal, through the load resistance, and back to V_{cc}. In this condition, the gate is said to be sinking current:



*Inverter gate **sinking** current*

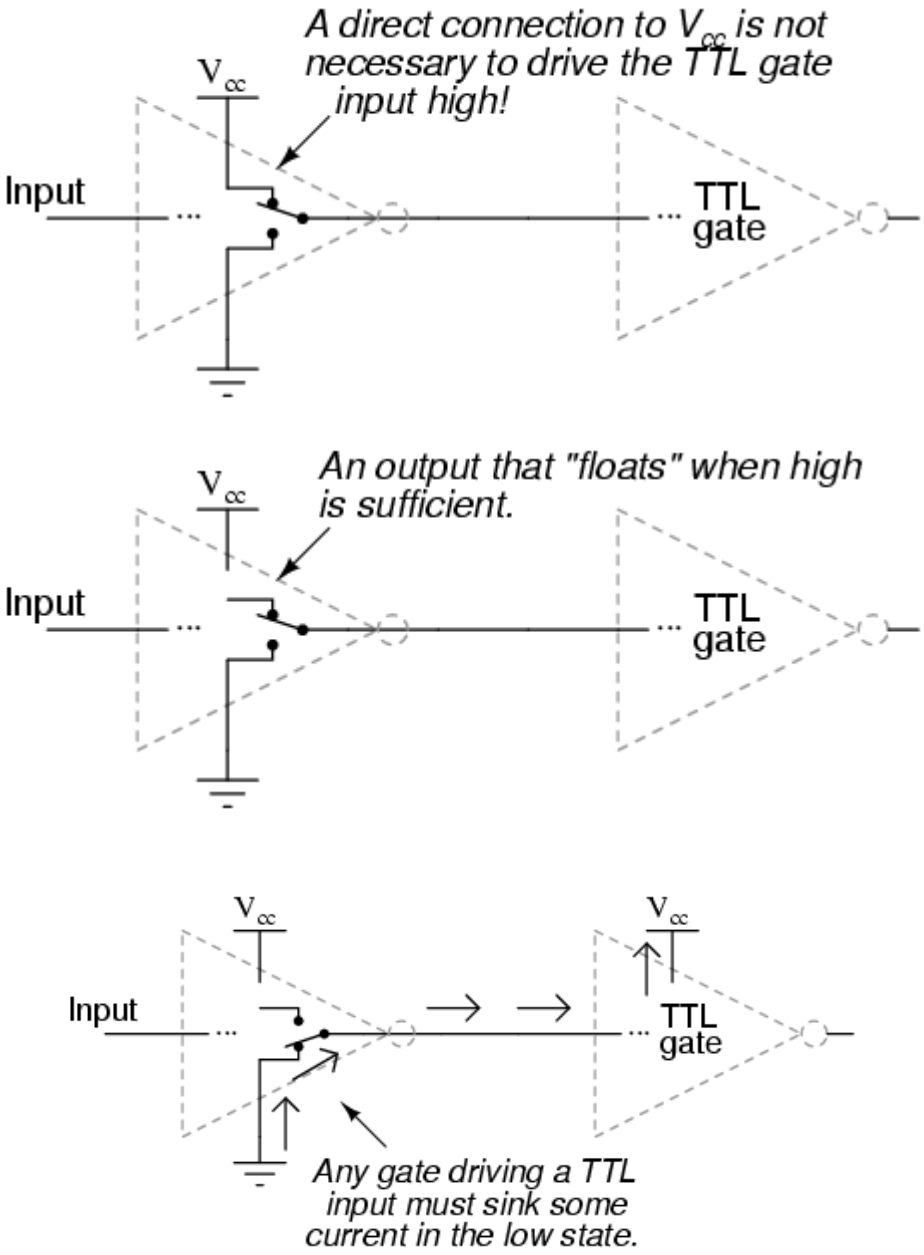
*Simplified gate circuit **sinking** current*



The combination of Q3 and Q4 working as a “push-pull” transistor pair (otherwise known as a totem pole output) has the ability to either source current (draw in current to V_{cc}) or sink current (output current from ground) to a load.

However, a standard TTL gate input never needs current to be sourced, *only sunk*.

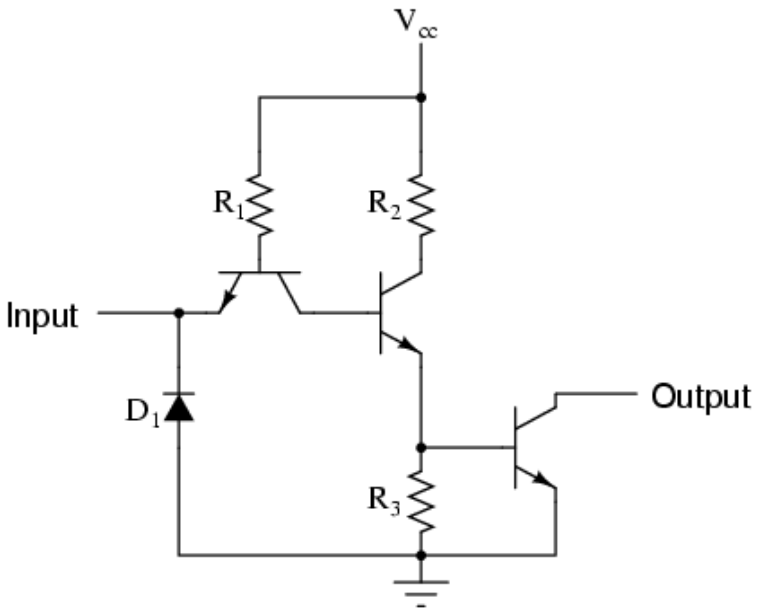
That is, since a TTL gate input naturally assumes a high state if left floating, any gate output driving a TTL input need only sink current to provide a “0” or “low” input, and need not source current to provide a “1” or a “high” logic level at the input of the receiving gate:



This means we have the option of simplifying the output stage of a gate circuit so as to eliminate Q3 altogether.

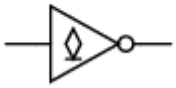
The result is known as an open-collector output:

Inverter circuit with open-collector output



To designate open-collector output circuitry within a standard gate symbol, a special marker is used. Shown here is the symbol for an inverter gate with open-collector output:

Inverter with open-collector output



Please keep in mind that the "high" default condition of a floating gate input is only true for TTL circuitry, and not necessarily for other types, especially for logic gates constructed of field-effect transistors.

Brief Recap:

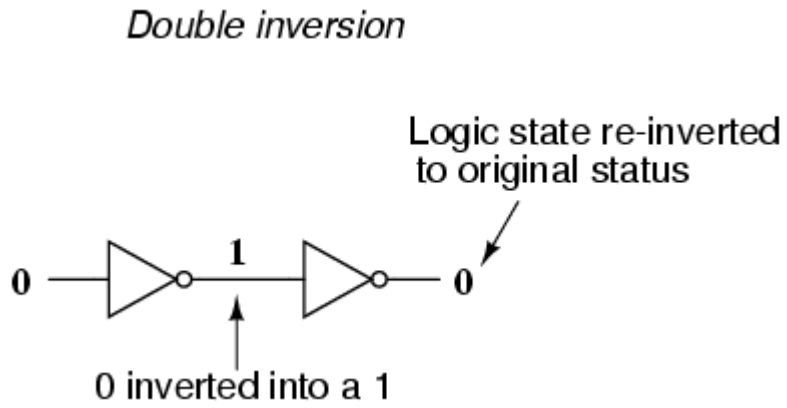
- An inverter (aka NOT gate) is one that outputs the opposite state as what is input. That is, a "low" input (0) gives a "high" output (1), and vice versa.
- Gate circuits constructed of resistors, diodes and bipolar transistors as illustrated in this section are called **TTL**. TTL is an acronym standing for **Transistor-to-Transistor Logic**. There are also other design methodologies used in gate circuits, such as with field-effect transistors (FET's), rather than bipolar transistors.
- A gate is said to be **sourcing current** when it provides a path for current between the *output terminal and the positive side of the DC power supply* (V_{cc}). In other words, it is connecting the output terminal to the power source (+V).

- A gate is said to be **sinking current** when it provides a path for current between the *output terminal and ground*. In other words, it is grounding (sinking) the output terminal.
- Gate circuits with **totem pole output stages** are able to *both source and sink current*.
- Gate circuits with **open-collector output stages** are *only able to sink current*, and not source current.
- Open-collector gates are practical when used to drive TTL gate inputs because TTL inputs don't require current sourcing.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-3/not-gate/

The "Buffer" Gate

If we were to connect two inverter gates together so that the output of one fed into the input of another, the two inversion functions would "cancel" each other out so that there would be no inversion from input to final output:

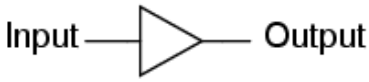


While this may seem like a pointless thing to do, it does have practical application. Remember that gate circuits are signal amplifiers, regardless of what logic function they may perform.

A weak signal source that is not capable of sourcing or sinking very much current to a load (such as with an Arduino) may be **boosted** by means of two inverters like the pair shown above. While the *logic level* is unchanged… the full current-sourcing/sinking capabilities of the final inverter are available to drive a load resistance if needed.

For this purpose, a special logic gate called a buffer is manufactured to perform the same function as two inverters. Its symbol is simply a triangle, with no inverting "bubble" on the output terminal:

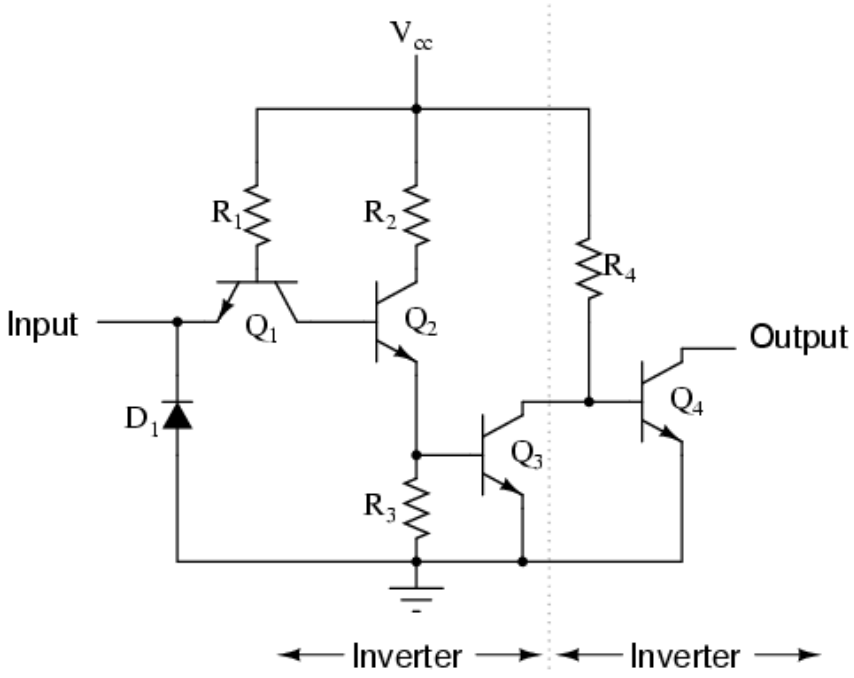
"Buffer" gate



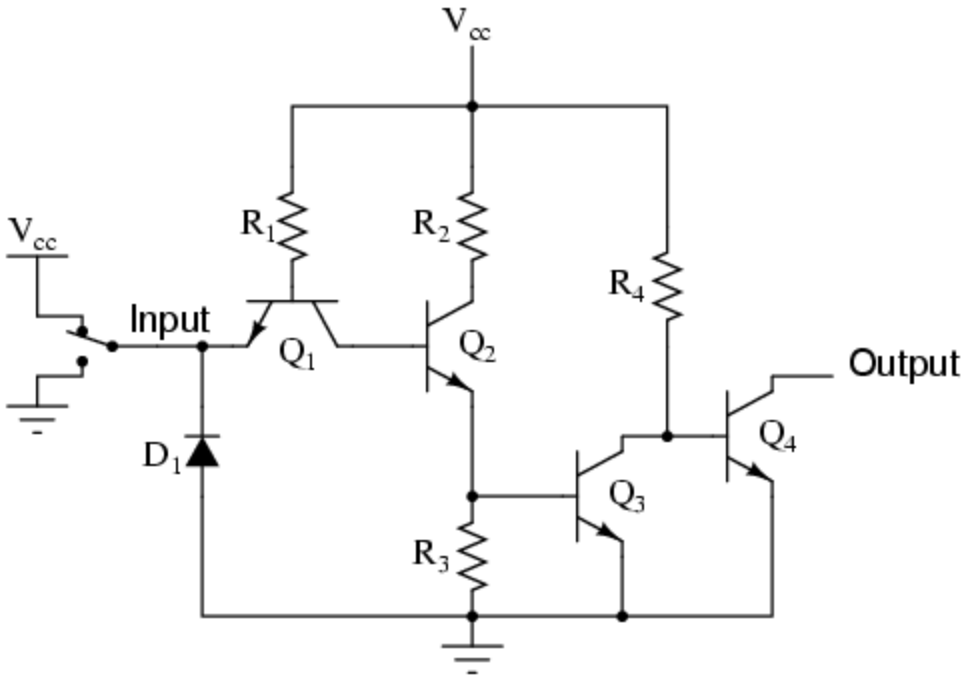
Input	Output
0	0
1	1

The internal schematic diagram for a typical open-collector buffer is not much different from that of a simple inverter: only one more common-emitter transistor stage is added to re-invert the output signal:

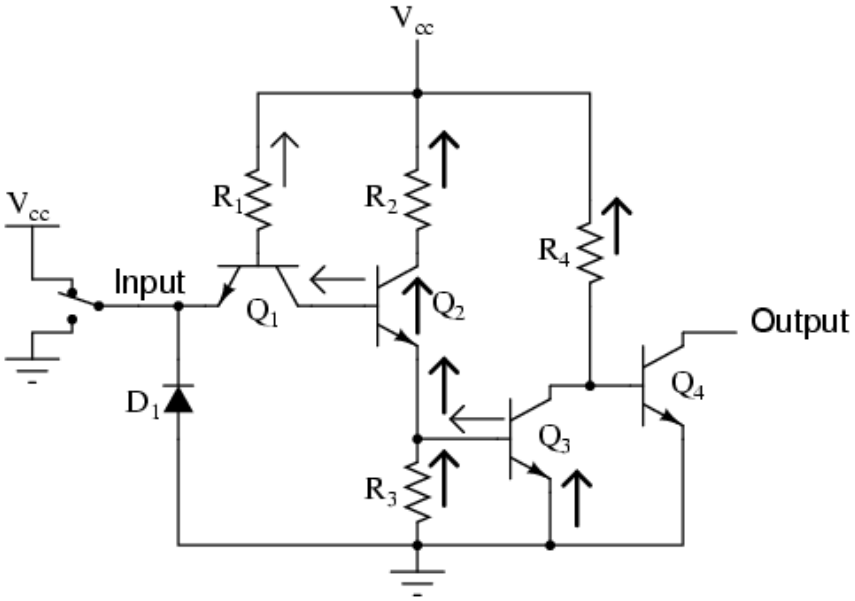
Buffer circuit with open-collector output



Let's analyze this circuit for two conditions: an input logic level of "1" and an input logic level of "0." First, a "high" (1) input:



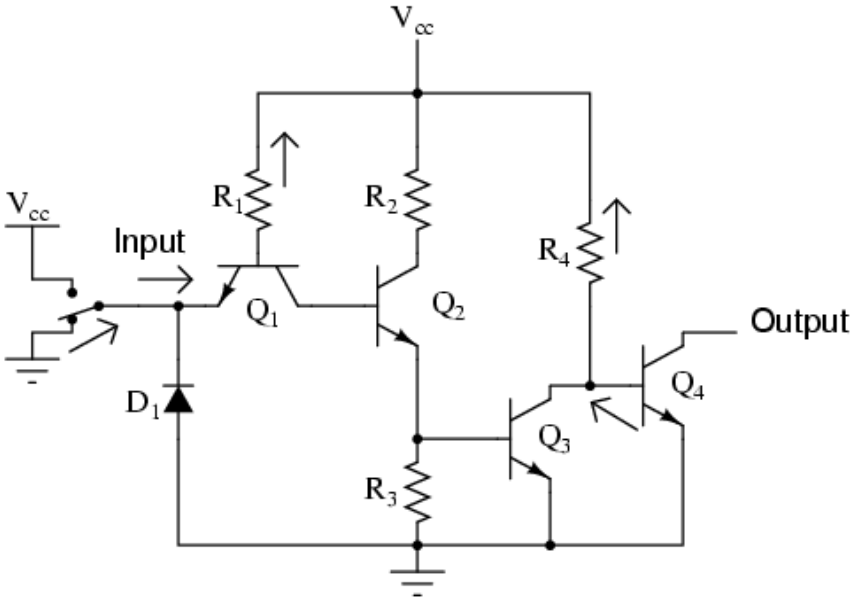
As before with the inverter circuit, the “high” input causes no conduction through the left steering diode of Q1 (emitter-to-base PN junction). All of R1‘s current goes through the base of transistor Q2, saturating it:



Having Q2 saturated causes Q3 to be saturated as well, resulting in very little voltage dropped between the base and emitter of the final output transistor Q4.

Thus, Q4 will be in cutoff mode, conducting no current. The output terminal will be floating (neither connected to ground nor Vcc), and this will be equivalent to a “high” state on the input of the next TTL gate that this one feeds in to. Thus, a “high” input gives a “high” output.

With a “low” input signal (input terminal grounded), the analysis looks something like this:



All of R1‘s current is now diverted through the input switch, thus eliminating base current through Q2.

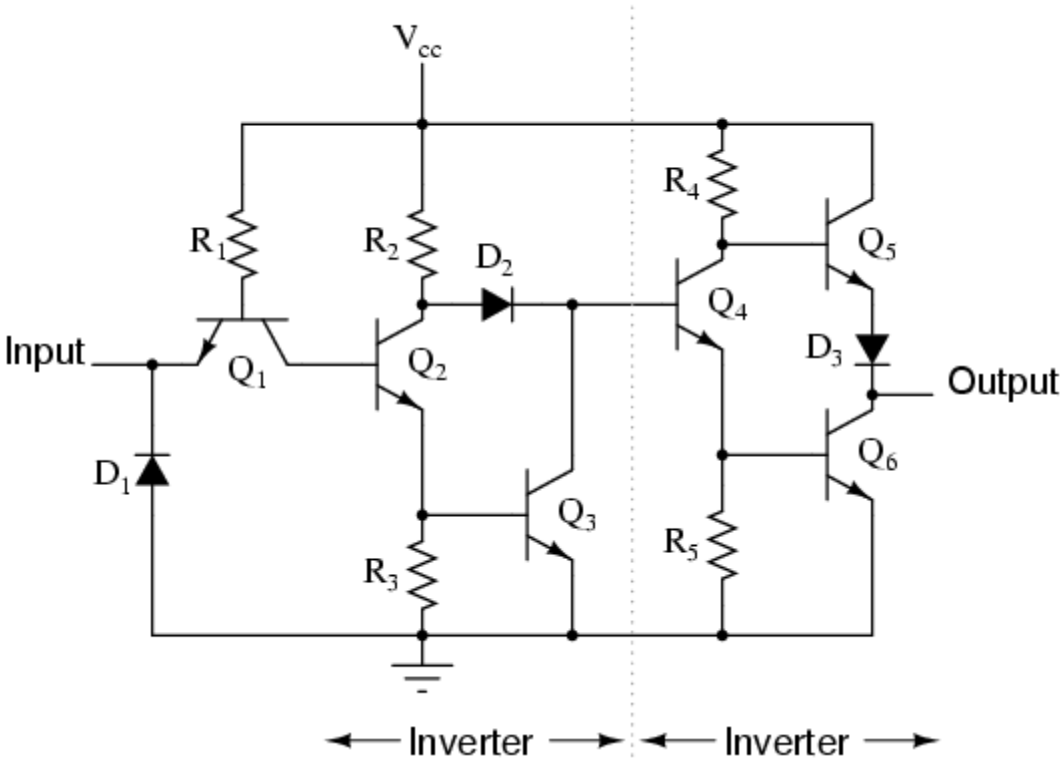
This forces transistor Q2 into cutoff so that no base current goes through Q3 either.

With Q3 cutoff as well, Q4 is will be saturated by the current through resistor R4, thus connecting the output terminal to ground, making it a “low” logic level.

Thus, a “low” input gives a “low” output.

The schematic diagram for a buffer circuit with totem pole output transistors is a bit more complex, but the basic principles, and certainly the truth table, are the same as for the open-collector circuit:

Buffer circuit with totem pole output



Brief Recap:

- Two inverters (i.e. NOT gates) connected in "series" so as to invert, then re-invert, a binary bit perform the function of a buffer.
- Buffer gates merely serve the purpose of signal amplification: taking a "weak" signal source that isn't capable of sourcing or sinking much current; whereby boosting the current capacity of the signal so as to be able to drive a load.
- Buffer circuits are symbolized by a triangle symbol with no inverter "bubble".
- Buffers, like inverters, may be made in open-collector output or totem pole output forms.

Multiple-Input Gates

Adding more input terminals to a logic gate increases the number of input state possibilities. With a single-input gate such as the inverter or buffer, there can only be two possible input states: either the input is "high" (1) or it is "low" (0).

Whereas a *two input gate* has **four possibilities** (00, 01, 10, and 11), and a *three-input gate* has **eight possibilities** (000, 001, 010, 011, 100, 101, 110, and 111) for input states, and so

forth. The number of possible input states is equal to two to the power of the number of inputs:

$$\text{Number of possible input states} = 2^n$$

Where,

n = Number of inputs

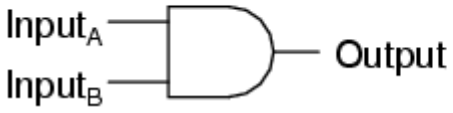
This increase in the number of possible input states obviously allows for more complex gate behavior. Now, instead of merely inverting or amplifying (buffering) a single "high" or "low" logic level, the output of the gate will be determined by whatever combination of 1's and 0's is present at the inputs.

Each of the main multi-input gate types will be presented in this section; showing its standard symbol, truth table, and practical operation.

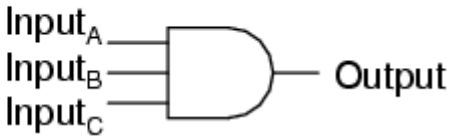
The AND Gate

One of the easiest multiple-input gates to understand is the AND gate, so-called because the output of this gate will be "high" (1) if and only if all inputs (first input and the second input and . . .) are "high" (1). If any input(s) is "low" (0), the output is guaranteed to be in a "low" state as well.

2-input AND gate

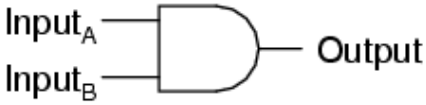


3-input AND gate



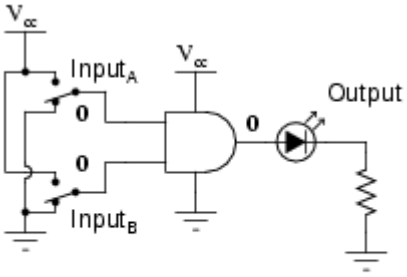
In case you might have been wondering, AND gates are made with more than three inputs, but this is less common than the simple two-input variety. A two-input AND gate's truth table looks like this:

2-input AND gate

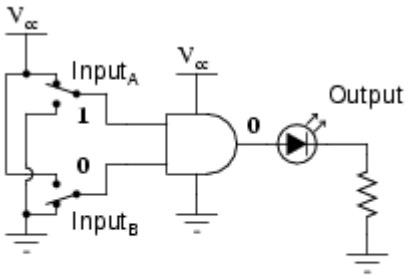


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

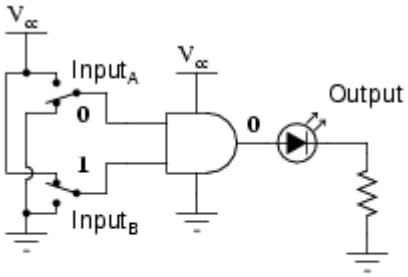
What this truth table means in practical terms is shown in the following sequence of illustrations, with the 2-input AND gate subjected to all possibilities of input logic levels. An LED (Light-Emitting Diode) provides visual indication of the output logic level:



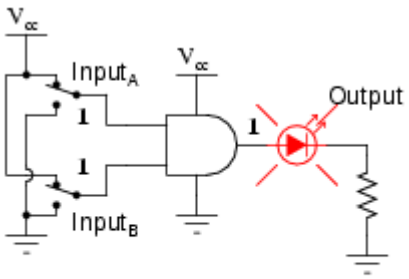
$Input_A$ = **0**
$Input_B$ = **0**
Output = **0** *(no light)*



$Input_A$ = **1**
$Input_B$ = **0**
Output = **0** *(no light)*



$Input_A$ = **0**
$Input_B$ = **1**
Output = **0** *(no light)*



$Input_A$ = **1**
$Input_B$ = **1**
Output = **1** *(light!)*

It is only with all inputs raised to “high” logic levels that the AND gate’s output goes “high,” thus energizing the LED for only one out of the four input combination states.

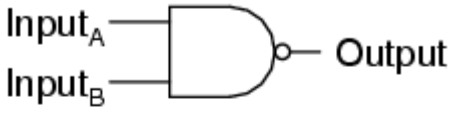
The NAND Gate

A variation on the idea of the AND gate is called the NAND gate. The word “NAND” is a verbal contraction of the words NOT and AND.

Essentially, a NAND gate behaves the same as an AND gate with a NOT (inverter) gate *connected to the output terminal*.

To symbolize this output signal inversion, the NAND gate symbol has a bubble on the output line. The truth table for a NAND gate is as one might expect, exactly opposite as that of an AND gate:

2-input NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Equivalent gate circuit

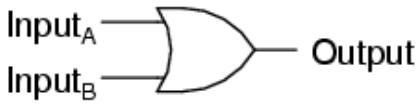
As with AND gates, NAND gates are made with more than two inputs. In such cases, the same general principle applies: the output will be “low” (0) if and only if all inputs are “high” (1).

If any input is “low” (0), the output will go “high” (1).

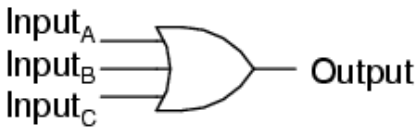
The OR Gate

Our next gate to investigate is the OR gate, so-called because the output of this gate will be “high” (1) if any of the inputs (first input or the second input or . . .) are “high” (1). The output of an OR gate goes “low” (0) if and only if all inputs are “low” (0).

2-input OR gate

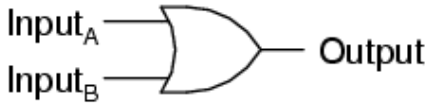


3-input OR gate



A two-input OR gate's truth table looks like this:

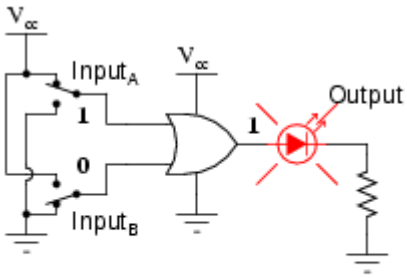
2-input OR gate



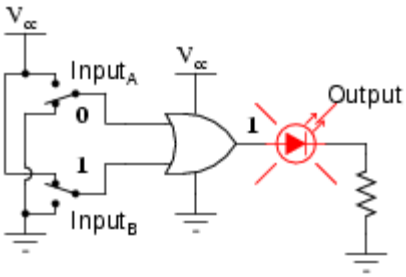
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

The following sequence of four illustrations demonstrates the OR gate's function; with the 2-inputs experiencing all possible logic levels.

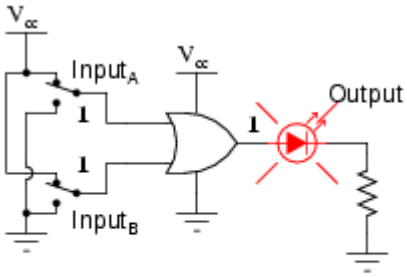
An LED (Light-Emitting Diode) provides visual indication of the gate's output logic level:



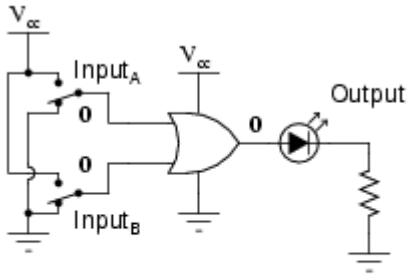
Input$_A$ = **1**
Input$_B$ = **0**
Output = **1** *(light!)*



Input$_A$ = **0**
Input$_B$ = **1**
Output = **1** *(light!)*



Input$_A$ = **1**
Input$_B$ = **1**
Output = **1** *(light!)*



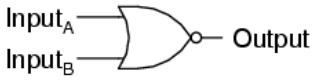
Input$_A$ = **0**
Input$_B$ = **0**
Output = **0** *(no light)*

A condition of any input being raised to a "high" logic level makes the OR gate's output go "high," thus energizing the LED for three out of the four input combination states.

The NOR Gate

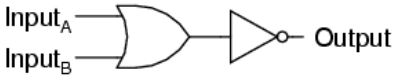
As you might have suspected, the NOR gate is an OR gate with its output inverted, just like a NAND gate is an AND gate with an inverted output:

2-input NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Equivalent gate circuit



NOR gates, like all the other multiple-input gates seen thus far, can be manufactured with more than two inputs. Still, the same logical principle applies: the output goes “low” (0) if any of the inputs are made “high” (1). The output is (1) only when all inputs are “low” (0).

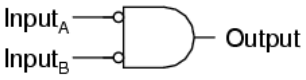
The Negative-AND Gate

A Negative-AND gate functions the same as an AND gate with all its inputs inverted (connected through NOT gates).

In keeping with standard gate symbol convention, these inverted inputs are signified by bubbles.

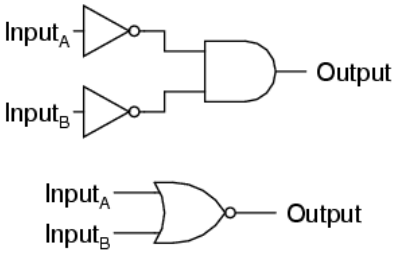
Contrary to most peoples’ first instinct, the logical behavior of a Negative-AND gate is not the same as a NAND gate; its truth table, actually, is identical to a NOR gate:

2-input Negative-AND gate



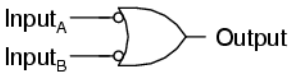
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Equivalent gate circuits



The Negative-OR Gate

2-input Negative-OR gate



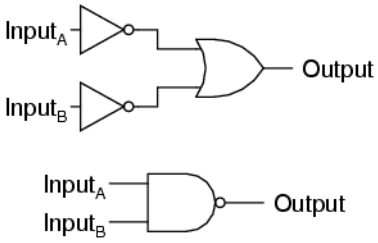
Following the same pattern, a Negative-OR gate functions the same as an OR gate with all its inputs inverted.

In keeping with standard gate symbol convention, these inverted inputs are signified by bubbles.

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

The behavior and truth table of a Negative-OR gate is the same as for a NAND gate:

Equivalent gate circuits



The Exclusive-OR Gate

Exclusive-OR gate

The last six gate types are all fairly direct variations on three basic functions: AND, OR, and NOT. The Exclusive-OR gate, however, is something quite different.

Exclusive-OR gates output a “high” (1) logic level if the inputs are at different logic levels, either 0 and 1 or 1 and 0.

Conversely, they output a “low” (0) logic level if the inputs are at the same logic levels.

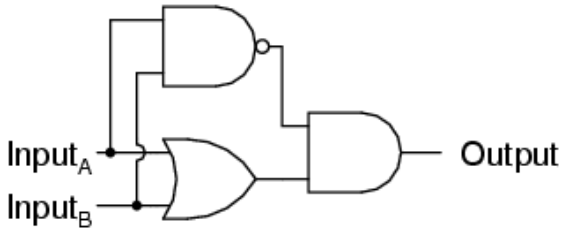
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

The Exclusive-OR (sometimes called **XOR**) gate has both a symbol and a truth table pattern that is unique:

There are equivalent circuits for an Exclusive-OR gate made up of AND, OR, and NOT gates, just as there were for NAND, NOR, and the negative-input gates.

A rather direct approach to simulating an Exclusive-OR gate is to start with a regular OR gate, then add additional gates to inhibit the output from going “high” (1) when both inputs are “high” (1):

Exclusive-OR equivalent circuit

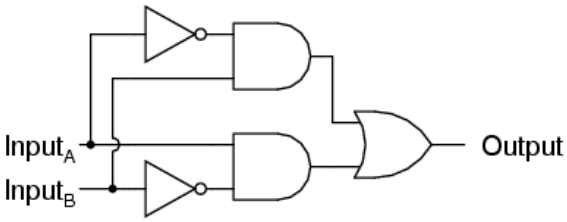


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In this circuit, the final AND gate act as a buffer for the output of the OR gate whenever the NAND gate's output is high, which it is for the first three input state combinations (00, 01, and 10). However, when both inputs are "high" (1), the NAND gate outputs a "low" (0) logic level, which forces the final AND gate to produce a "low" (0) output.

Another equivalent circuit for the Exclusive-OR gate uses a strategy of two AND gates with inverters, set up to generate "high" (1) outputs for input conditions 01 and 10. A final OR gate then allows either of the AND gates' "high" outputs to create a final "high" output:

Exclusive-OR equivalent circuit

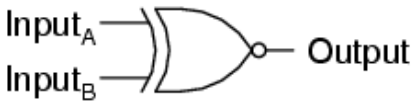


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-OR gates are very useful for circuits where *two or more binary numbers are to be compared bit-for-bit*, and also for **error detection** (parity check) and **code conversion** (binary to Grey and vice versa).

The Exclusive-NOR Gate

Exclusive-NOR gate



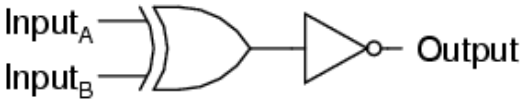
Finally, our last gate for analysis is the Exclusive-NOR gate, otherwise known as the XNOR gate.

It is equivalent to an Exclusive-OR gate with an inverted output. The truth table for this gate is exactly opposite as for the Exclusive-OR gate:

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

As indicated by the truth table, the purpose of an Exclusive-NOR gate is to output a "high" (1) logic level whenever both inputs are at the same logic levels (either 00 or 11).

Equivalent gate circuit



Brief Recap:

- Rule for an AND gate: output is "high" only if first input and second input are both "high."
- Rule for an OR gate: output is "high" if input A or input B are "high."
- Rule for a NAND gate: output is not "high" if both the first input and the second input are "high."
- Rule for a NOR gate: output is not "high" if either the first input or the second input are "high."
- A Negative-AND gate behaves like a NOR gate.
- A Negative-OR gate behaves like a NAND gate.
- Rule for an Exclusive-OR gate: output is "high" if the input logic levels are different.
- Rule for an Exclusive-NOR gate: output is "high" if the input logic levels are the same.

Attribution: https://www.allaboutcircuits.com/textbook/digital/chpt-3/digital-signals-gates/

Part 3: Handy Tables, Charts & Formulas

Ohm's Law

If we only know voltage (E) and resistance (R):

$$\textit{If,} \quad I = \frac{E}{R} \quad \textit{and} \quad P = I E$$

$$\textit{Then,} \quad P = \frac{E}{R} E \quad \textit{or} \quad \mathbf{P = \frac{E^2}{R}}$$

If we only know current (I) and resistance (R):

$$\textit{If,} \quad E = I R \quad \textit{and} \quad P = I E$$

$$\textit{Then,} \quad P = I(I R) \quad \textit{or} \quad \mathbf{P = I^2 R}$$

Trivia: Contrary to popular belief, It was **James Prescott Joule**, not Georg Simon Ohm, who first discovered the mathematical relationship between power dissipation and current through a resistance.

This discovery, published in 1841, followed the form of the last equation (P = I 2 R), and is properly known as Joule's Law. However, these power equations are so commonly associated with the Ohm's Law equations relating voltage, current, and resistance (E=IR ; I=E/R ; and R=E/I) that they are frequently credited to Ohm.

Power equations

$$P = IE \qquad P = \frac{E^2}{R} \qquad P = I^2 R$$

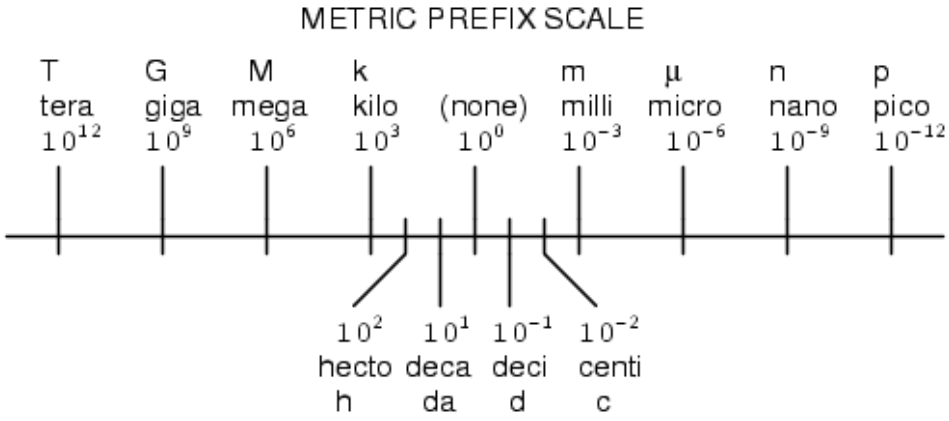
Attribution: http://www.allaboutcircuits.com/vol_1/chpt_2/4.html

Kirchhoff's Law

- **Kirchhoff's Voltage Law (KVL) Summarized:** "The algebraic sum of all voltages in a loop must equal zero"
- **Kirchhoff's Current Law (KCL) Summarized:** "The algebraic sum of all currents entering and exiting a node must equal zero"

Misc. Electrical Conversions:

- 1 Horsepower = 745.7 watts
- 1 Amp = 6.25 x 10 18 electrons per second
- 1 Ah/1000 mAh (amp-hour) = 3600 coulombs
- Proton Mass = 1.67 x 10 -24 grams



POWER	METRIC PREFIX
12	Tera (T)
9	Giga (G)
6	Mega (M)
3	Kilo (k)
0	UNITS (plain)
-3	milli (m)
-6	micro (u)
-9	nano (n)
-12	pico (p)

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_4/3.html

Scientific Notation Basics

To enter numbers in scientific notation into a hand calculator, there is usually a button marked "E" or "EE" used to enter the correct power of ten. For example, to enter the mass of a proton in grams (1.67 x 10 -24 grams) into a calculator, enter the following (minus the brackets):

```
[1]   [.]   [6]   [7]   [EE]   [2]   [4]   [+/-]
```

The [+/-] keystroke changes the sign of the power (24) into a -24. Some calculators allow the use of the subtraction key [-] to do this, but

I prefer the "change sign" [+/-] key because its more consistent with the use of that key in other contexts.

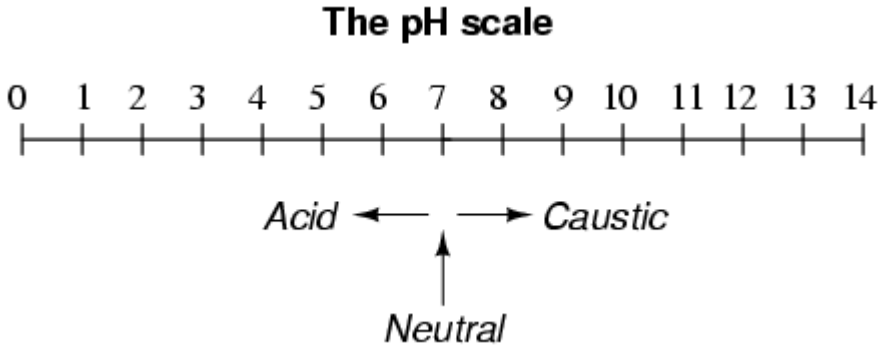
If I wanted to enter a negative number in scientific notation into a hand calculator, I would have to be careful how I used the [+/-] key, lest I change the sign of the power and not the significant digit value.

Example: Number to be entered: -3.221 x 10 -15 :

```
[3]   [.]   [2]   [2]   [1]   [+/-]   [EE]   [1]   [5]   [+/-]
```

The first [+/-] keystroke changes the entry from 3.221 to -3.221; the second [+/-] keystroke changes the power from 15 to -15.

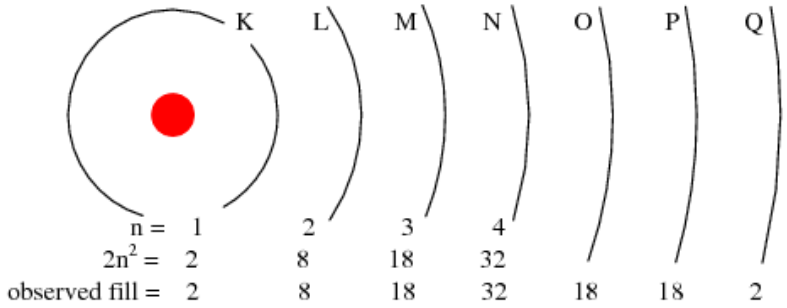
Attribution: http://www.allaboutcircuits.com/vol_1/chpt_4/5.html



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_9/6.html

Electron Basics:

The maximum number of electrons that any shell may hold is described by the equation 2n 2 , where “n” is the principle quantum number. Thusly, the first shell (n=1) can hold 2 electrons; the second shell (n=2) 8 electrons, and the third shell (n=3) 18 electrons:



Principal quantum number n and maximum number of electrons per shell both predicted by 2(n 2), and observed. Orbitals not to scale.

Electron shells in an atom were formerly designated by letter rather than by number. The first shell (n=1) was labeled K, the second shell (n=2) L, the third shell (n=3) M, the fourth shell (n=4) N, the fifth shell (n=5) O, the sixth shell (n=6) P, and the seventh shell (n=7) Q.

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/2.html

Body Resistance Chart:

BODILY EFFECT	DIRECT CURRENT (DC)	60 Hz AC	10 kHz AC
Slight sensation felt at hand(s)	Men = 1.0 mA Women = 0.6 mA	0.4 mA 0.3 mA	7 mA 5 mA
Threshold of perception	Men = 5.2 mA Women = 3.5 mA	1.1 mA 0.7 mA	12 mA 8 mA
Painful, but voluntary muscle control maintained	Men = 62 mA Women = 41 mA	9 mA 6 mA	55 mA 37 mA
Painful, unable to let go of wires	Men = 76 mA Women = 51 mA	16 mA 10.5 mA	75 mA 50 mA
Severe pain, difficulty breathing	Men = 90 mA Women = 60 mA	23 mA 15 mA	94 mA 63 mA
Possible heart fibrillation after 3 seconds	Men = 500 mA Women = 500 mA	100 mA 100 mA	

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_3/4.html

Solid, Round Copper Conductor Wire Table

Size AWG	Diameter inches	Cross-sectional area cir. mils	Cross-sectional area sq. inches	Weight lb/1000 ft
4/0	0.4600	211,600	0.1662	640.5
3/0	0.4096	167,800	0.1318	507.9
2/0	0.3648	133,100	0.1045	402.8
1/0	0.3249	105,500	0.08289	319.5
1	0.2893	83,690	0.06573	253.5
2	0.2576	66,370	0.05213	200.9
3	0.2294	52,630	0.04134	159.3
4	0.2043	41,740	0.03278	126.4
5	0.1819	33,100	0.02600	100.2
6	0.1620	26,250	0.02062	79.46
7	0.1443	20,820	0.01635	63.02
8	0.1285	16,510	0.01297	49.97
9	0.1144	13,090	0.01028	39.63
10	0.1019	10,380	0.008155	31.43
11	0.09074	8,234	0.006467	24.92
12	0.08081	6,530	0.005129	19.77
13	0.07196	5,178	0.004067	15.68
14	0.06408	4,107	0.003225	12.43
15	0.05707	3,257	0.002558	9.858
16	0.05082	2,583	0.002028	7.818
17	0.04526	2,048	0.001609	6.200
18	0.04030	1,624	0.001276	4.917
19	0.03589	1,288	0.001012	3.899
20	0.03196	1,022	0.0008023	3.092
21	0.02846	810.1	0.0006363	2.452
22	0.02535	642.5	0.0005046	1.945
23	0.02257	509.5	0.0004001	1.542
24	0.02010	404.0	0.0003173	1.233
25	0.01790	320.4	0.0002517	0.9699
26	0.01594	254.1	0.0001996	0.7692
27	0.01420	201.5	0.0001583	0.6100
28	0.01264	159.8	0.0001255	0.4837
29	0.01126	126.7	0.00009954	0.3836
30	0.01003	100.5	0.00007894	0.3042
31	0.008928	79.70	0.00006260	0.2413
32	0.007950	63.21	0.00004964	0.1913
33	0.007080	50.13	0.00003937	0.1517
34	0.006305	39.75	0.00003122	0.1203
35	0.005615	31.52	0.00002476	0.09542
36	0.005000	25.00	0.00001963	0.07567
37	0.004453	19.83	0.00001557	0.06001
38	0.003965	15.72	0.00001235	0.04759
39	0.003531	12.47	0.000009793	0.03774
40	0.003145	9.888	0.000007766	0.02993
41	0.002800	7.842	0.000006159	0.02374
42	0.002494	6.219	0.000004884	0.01882
43	0.002221	4.932	0.000003873	0.01493
44	0.001978	3.911	0.000003072	0.01184

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_12/2.html

The Bel & dB

Because the Bel was too large of a unit to be used directly, it became customary to apply the metric prefix deci (meaning 1/10) to it, making it decibels, or dB. Now, the expression "dB" is so common that many people do not realize it is a combination of "deci-" and "-Bel," or that there even is such a unit as the "Bel."

To put this into perspective, here is a table contrasting power gain/loss ratios against decibels:

Table: Gain / loss in decibels

Loss/gain as a ratio	Loss/gain in decibels	Loss/gain as a ratio	Loss/gain in decibels
$\frac{P_{output}}{P_{input}}$	$10 \log \frac{P_{output}}{P_{input}}$	$\frac{P_{output}}{P_{input}}$	$10 \log \frac{P_{output}}{P_{input}}$
1000	30 dB	0.1	-10 dB
100	20 dB	0.01	-20 dB
10	10 dB	0.001	-30 dB
1 (no loss or gain)	0 dB	0.0001	-40 dB

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_1/5.html

Trivia: An adaptation of the dBm scale for audio signal strength is used in studio recording and broadcast engineering for standardizing volume levels, and is called the **VU scale**.

VU meters are frequently seen on electronic recording instruments to indicate whether or not the recorded signal exceeds the maximum signal level limit of the device, where significant distortion will occur.

This "signal indicator" scale is calibrated in according to the dBm scale; albeit it does not directly indicate dBm for any signal other than *steady sine-wave tones*. The proper unit of measurement for a VU meter is **volume units**.

Table: Absolute power levels in dBm (decibel milliwatt)

Power in watts	Power in milliwatts	Power in dBm	Power in milliwatts	Power in dBm
1	1000	30 dB	1	0 dB
0.1	100	20 dB	0.1	-10 dB
0.01	10	10 dB	0.01	-20 dB
0.004	4	6 dB	0.001	-30 dB
0.002	2	3 dB	0.0001	-40 dB

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_1/6.html

Resistor Essentials

A resistor is a passive two-terminal electrical component that creates electrical resistance within a circuit. The ohm (symbol: Ω) is the global numeric Greek symbol of electrical resistance:

Resistor Schematic Symbol:

Resistor Numeric Symbol:

Ω

Resistors are used to control/limit the amount of current in a circuit (or to provide a specific voltage drop). Here's a typical 330Ω resistor:

Attribution: http://www.electronicspectrum.com/2012/03/resistor.html

Most experimenter-grade resistors have *four bands* – three color bands that determine its value (digit, digit, multiplier), and a fourth band for its **tolerance** (the acceptable +/- % deviation from the absolute value of the resistor). However, resistors with *only three bands* have a +/-20% tolerance.

The tolerance band helps you to determine whether the resistor is within acceptable limits, or if it should be tossed out.

To help you determine which band is the first, here's some handy guidelines:

- Some resistors have the color bands grouped/close to one end. Hold the resistor with the closely grouped bands to your left and read the resistor from left to the right.

- With 5% and 10% resistors (most common), the procedure is simple: Hold the resistor with the silver or gold band to the right and read the resistor from the left to the right. Since the first band can't be silver or gold, you'll know instantly where to start.

Resistor Color Code Chart:

The standard resistor color code table:

Color	Digit 1	Digit 2	Digit 3*	Multiplier	Tolerance	Temp. Coef.	Fail Rate
Black	0	0	0	$\times 10^0$			
Brown	1	1	1	$\times 10^1$	±1% (F)	100 ppm/K	1%
Red	2	2	2	$\times 10^2$	±2% (G)	50 ppm/K	0.1%
Orange	3	3	3	$\times 10^3$		15 ppm/K	0.01%
Yellow	4	4	4	$\times 10^4$		25 ppm/K	0.001%
Green	5	5	5	$\times 10^5$	±0.5% (D)		
Blue	6	6	6	$\times 10^6$	±0.25% (C)		
Violet	7	7	7	$\times 10^7$	±0.1% (B)		
Gray	8	8	8	$\times 10^8$	±0.05% (A)		
White	9	9	9	$\times 10^9$			
Gold				×0.1	±5% (J)		
Silver				×0.01	±10% (K)		
None					±20% (M)		

** 3rd digit - only for 5-band resistors*

Tip: When determining suitable resistor **wattage**, a safe rule of thumb is to use resistors rated *at least twice the wattage* that your circuit requires for continuous use. Also, the following attribution link also contains a handy online color code calculator.

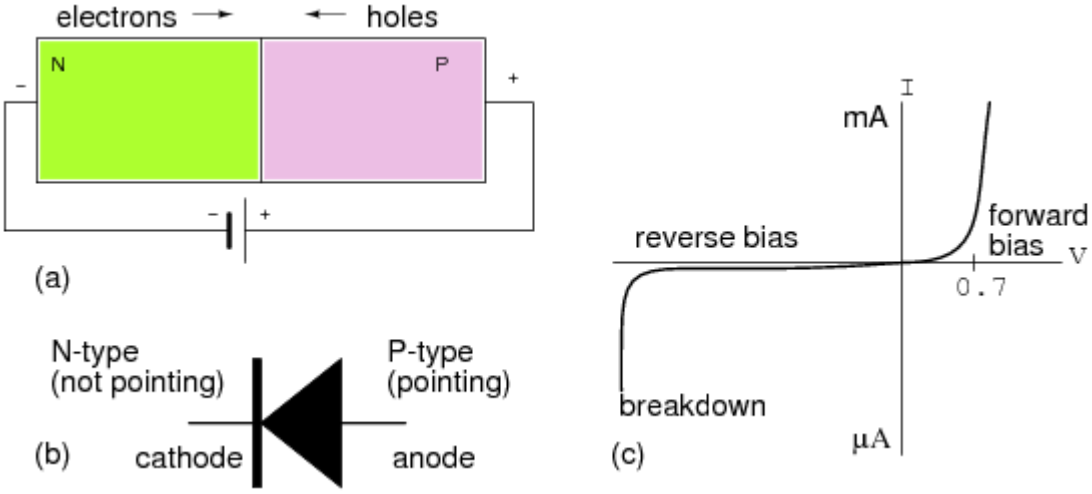
Attribution: http://www.hobby-hour.com/electronics/resistorcalculator.php

Semiconductor Essentials

Diode Basics

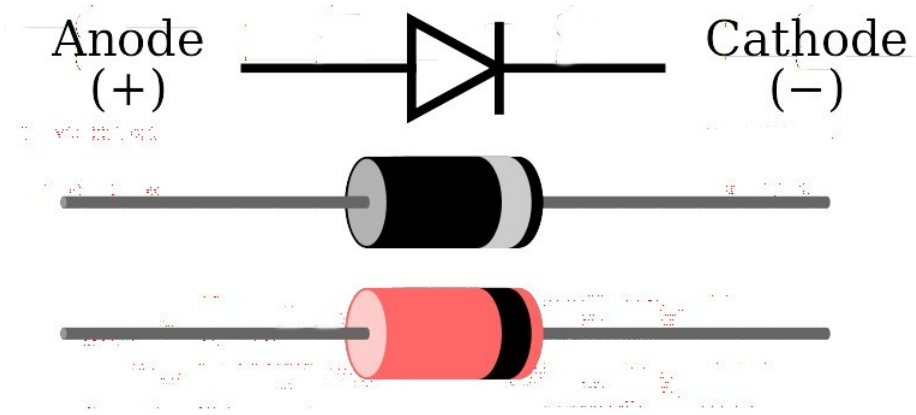
At the heart of the semiconductor family is the diode; a unidirectional device in which the cathode (bar) of the diode symbol corresponds to N-type semiconductor and the anode (arrow) corresponds to the P-type.

A typical silicon diode serves to pass voltage one direction, while blocking voltage in the opposite direction.



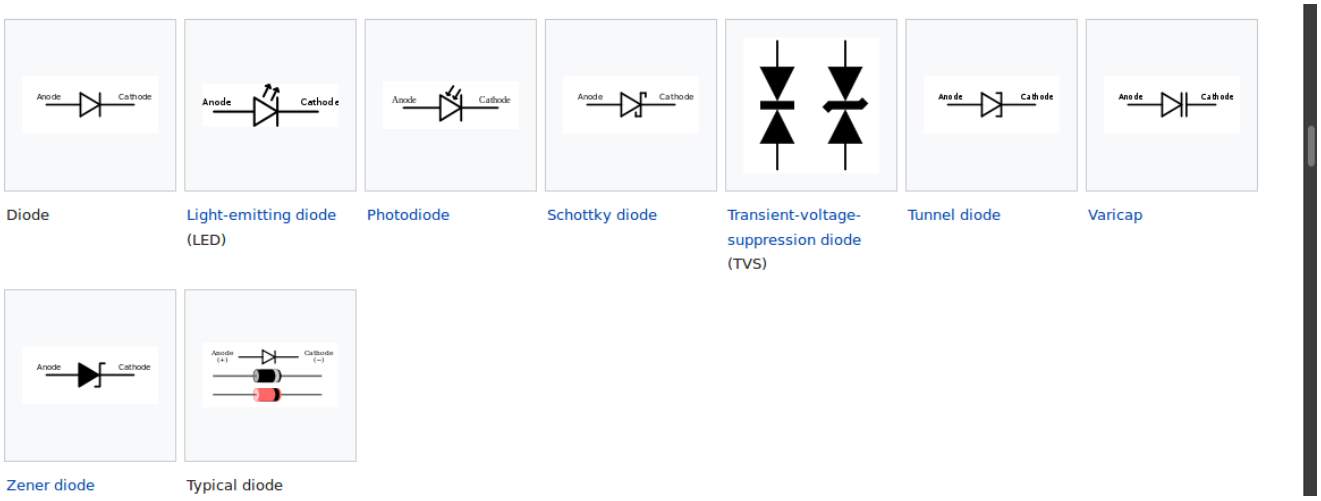
Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/6.html

To remember this relationship, Not-pointing (bar) on the symbol corresponds to N-type semiconductor. Pointing (arrow) corresponds to P-type:



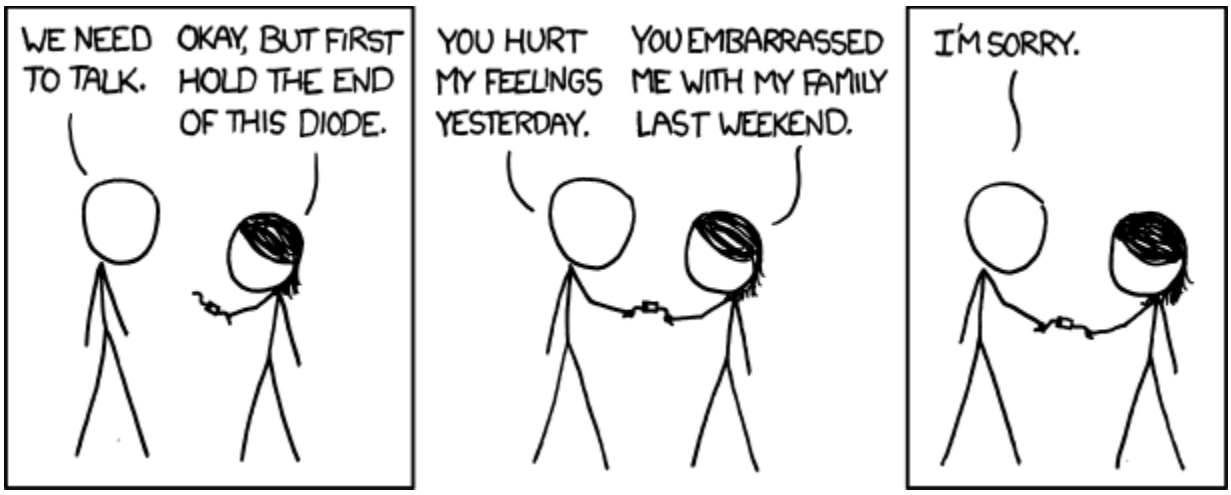
Attribution: http://en.wikipedia.org/wiki/File:Diode_pinout_en_fr.svg

Diode Symbol Examples:



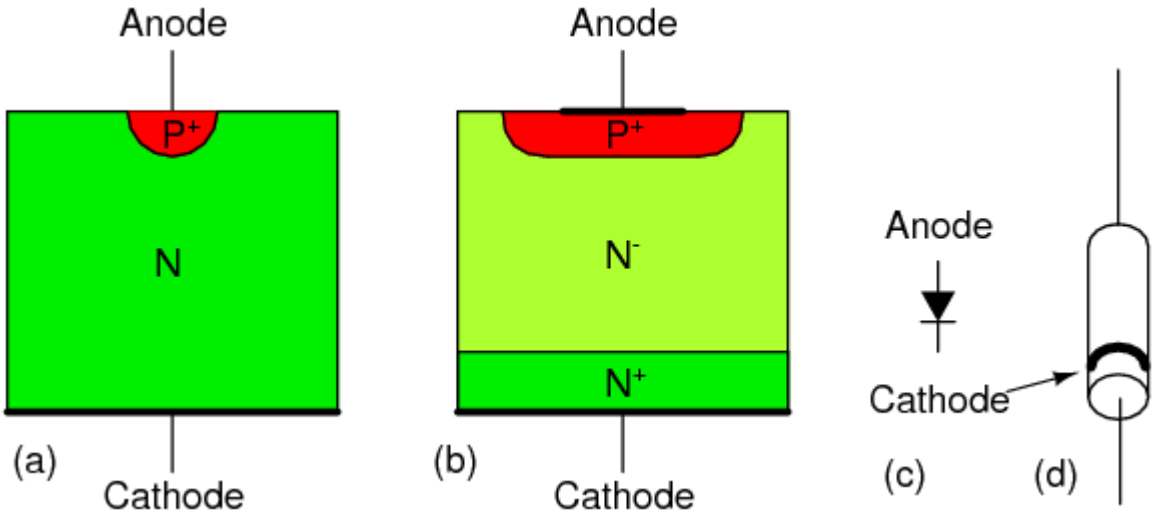
Attribution: http://en.wikipedia.org/wiki/Diode

Important Note: Each in-line silicon diode within a circuit has a typical **voltage drop of between 0.6-0.9 volts**. So be sure to factor that into your circuit designs. Also, it's best to use diodes rated *at least twice the minimum amperage and voltage* for your circuit.



Attribution: http://xkcd.com/814/

Transistor Basics

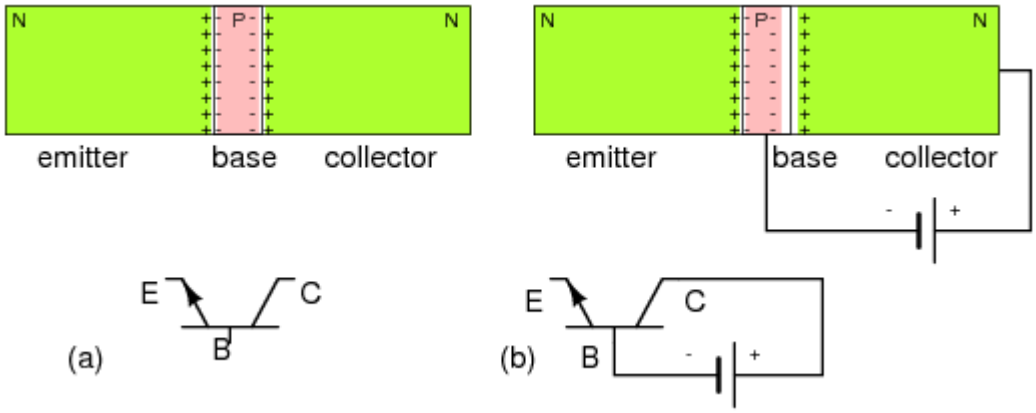


Silicon diode cross-section: (a) point contact diode, (b) junction diode, (c) schematic symbol, (d) small signal diode package.

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/7.html

In the figure below, fig(a) is an NPN junction bipolar transistor. Fig(b) represents applying reverse bias to collector base junction. Note that this increases the width of the depletion region. The reverse bias voltage could be a few volts to tens of volts for most transistors.

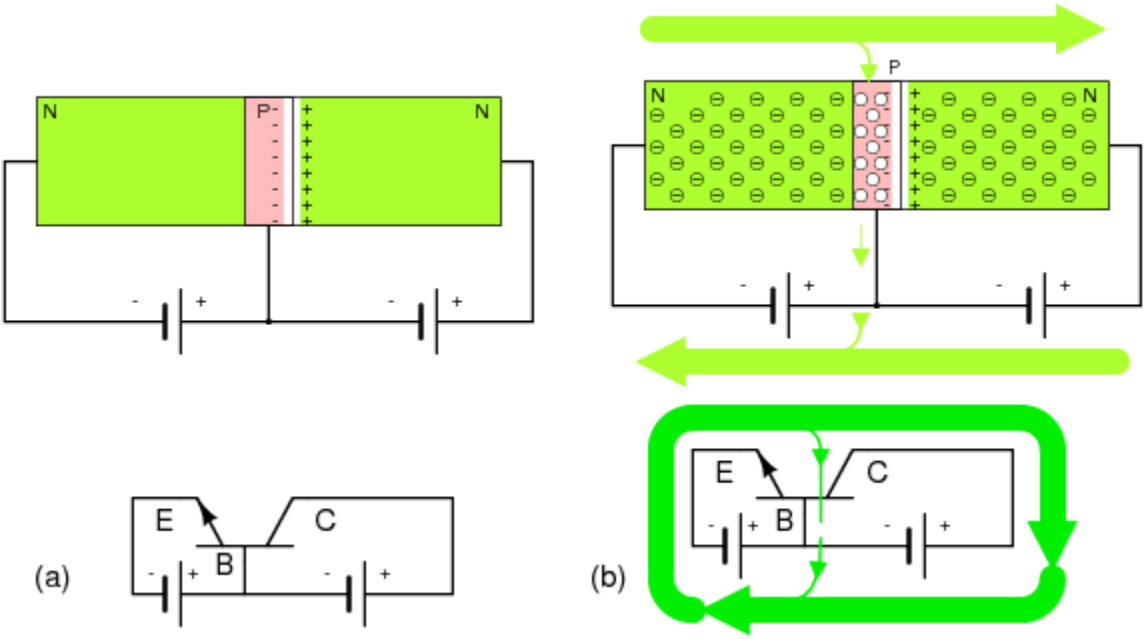
There is no current flow, except leakage current, in the collector circuit.



However, the base is manufactured thin. A few majority carriers in the emitter, injected as minority carriers into the base, actually recombine. See Fig(b) below.

Most of the emitter current of electrons diffuses through the thin base into the collector. Moreover, modulating the small base current produces a larger change in collector current. If the base voltage falls below approximately 0.6 V for a silicon transistor, the large

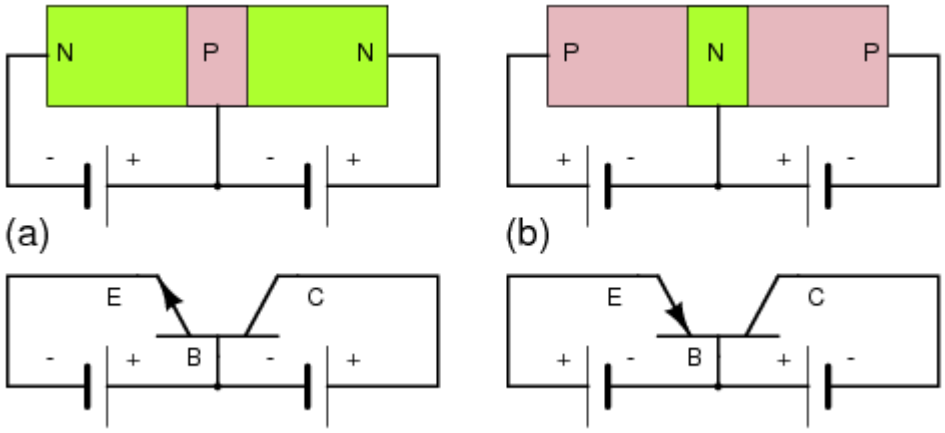
emitter-collector current ceases to flow.



NPN junction bipolar transistor with reverse biased collector-base:
(a) Adding forward bias to base-emitter junction, results in (b) a small base current and large emitter and collector currents.

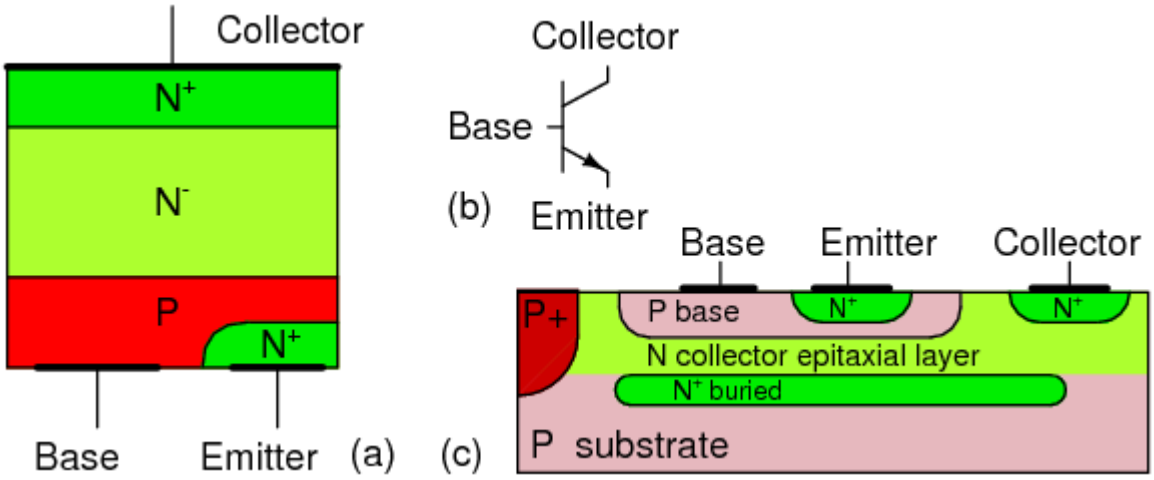
The difference between a PNP and NPN transistor is the polarity of the **base emitter diode junctions**, as signified by the direction of the schematic symbol emitter arrow. It points in the same direction *as the anode arrow for a junction diode* (i.e. against electron current flow).

However, the base-collector junction is the same polarity as the base-emitter junction, as compared to a diode. Compare NPN transistor at (a) with the PNP transistor at (b) below, noting direction of emitter arrow and supply polarity:

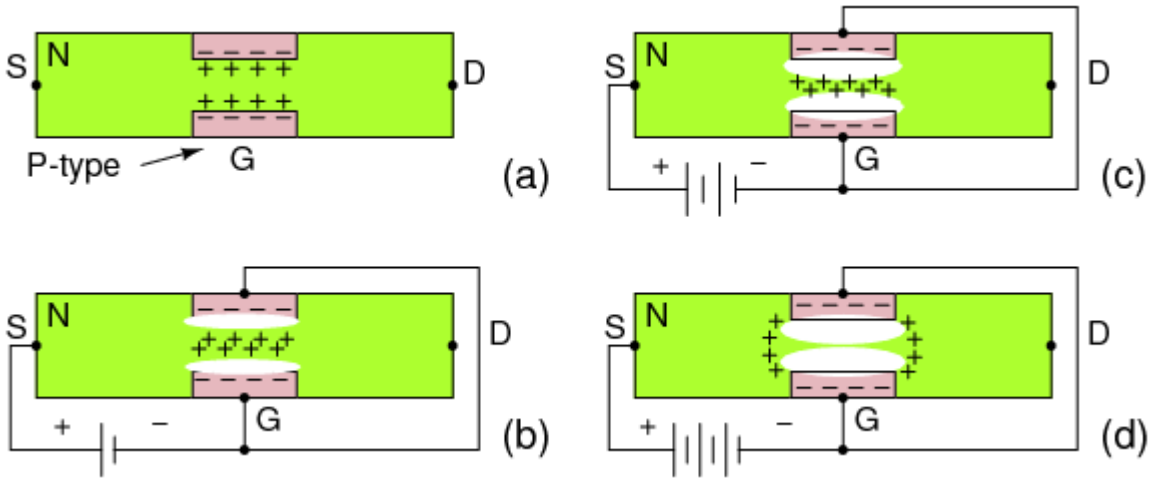


The voltage sources for PNP transistors are reversed compared with an NPN transistors as shown in above Figure. The base-emitter junction must be forward biased in both cases. The base on a PNP transistor is biased negative (b) compared with positive (a) for an NPN.

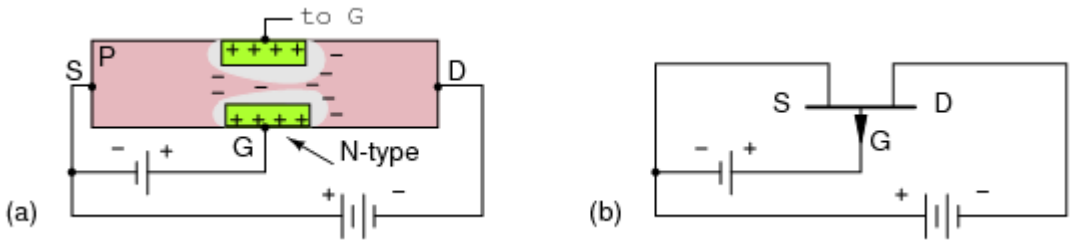
In both cases the base-collector junction is reverse biased. The PNP collector power supply is negative compared with positive for an NPN transistor.



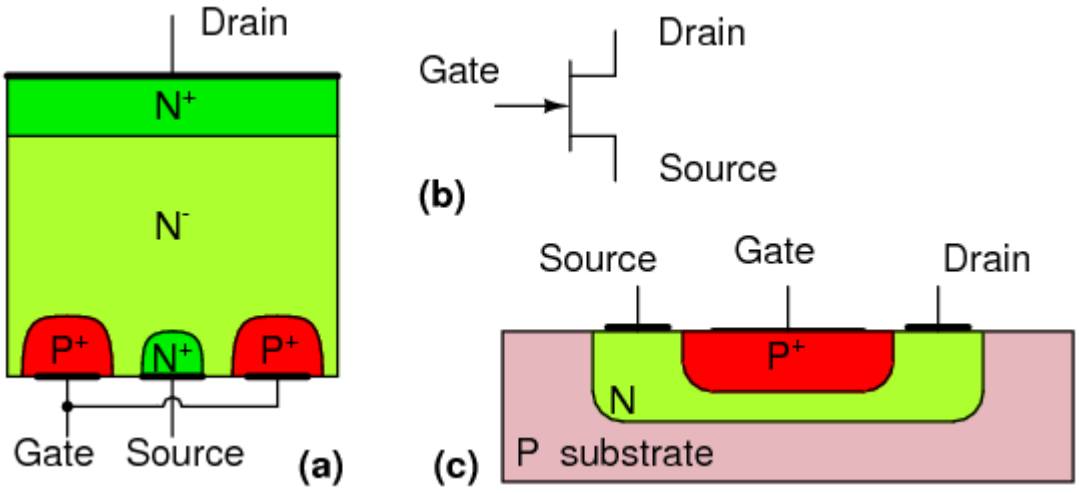
Bipolar junction transistor: (a) discrete device cross-section, (b) schematic symbol, (c) integrated circuit cross-section.



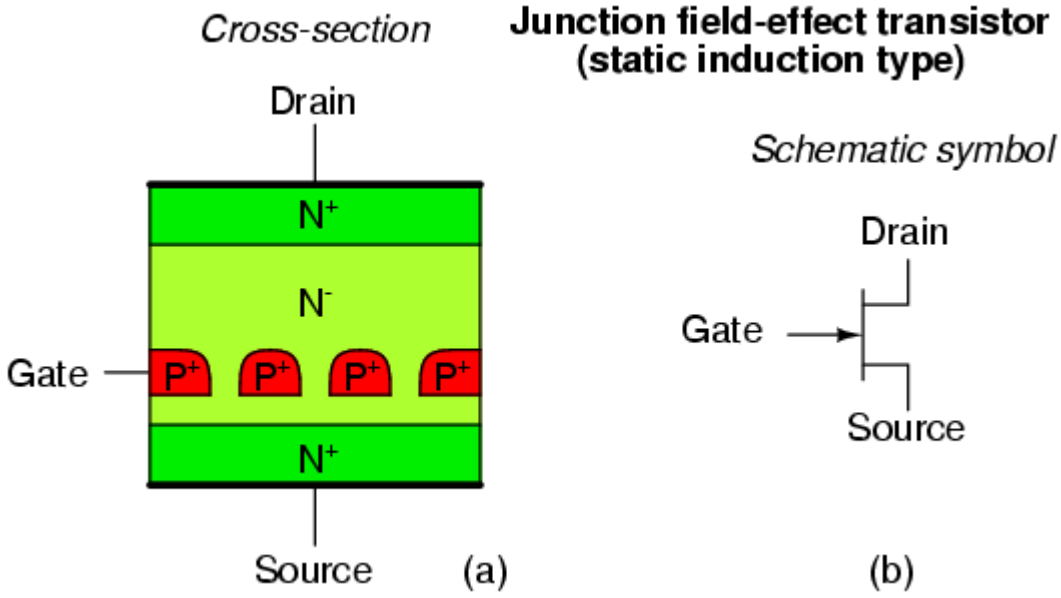
N-channel JFET: (a) Depletion at gate diode.
(b) Reverse biased gate diode increases depletion region.
(c) Increasing reverse bias enlarges depletion region.
(d) Increasing reverse bias pinches-off the S-D channel.



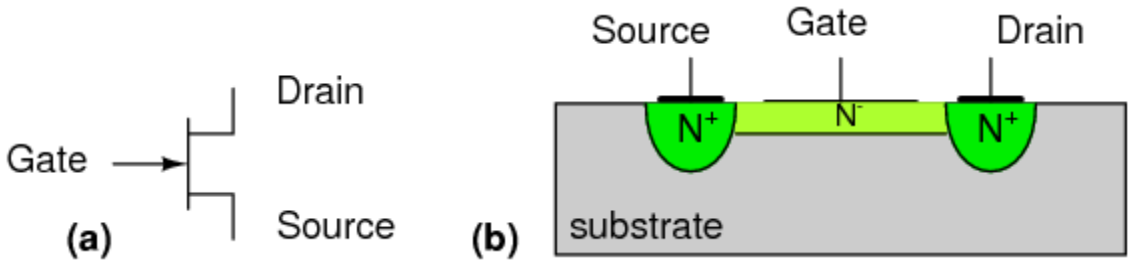
P-channel JFET: (a) N-type gate, P-type channel, reversed voltage sources compared with N-channel device. (b) Note reversed gate arrow/voltage sources on schematic.



Junction field effect transistor: (a) Discrete device cross-section, (b) schematic symbol, (c) integrated circuit device cross-section.

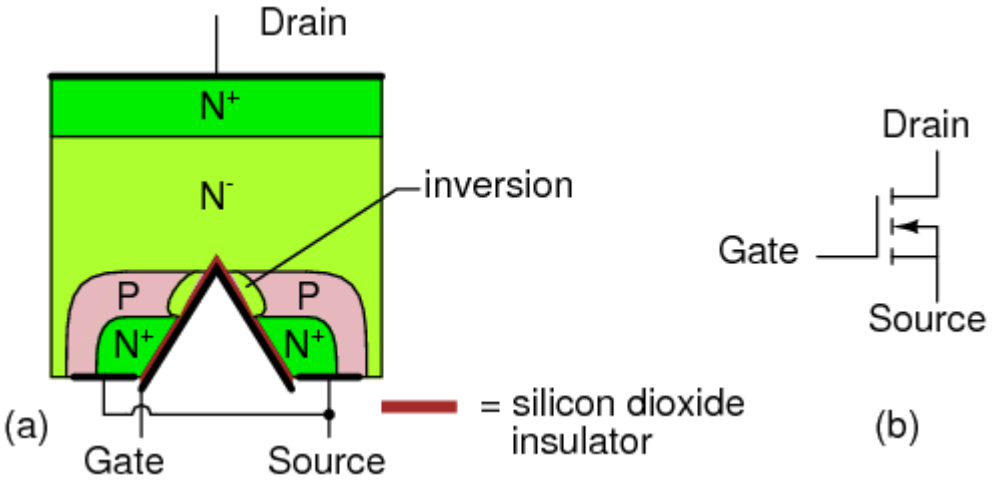


Junction field effect transistor (static induction type): (a) Cross-section, (b) schematic symbol.



The Metal semiconductor field effect transistor (MESFET) above is similar to a JFET except the gate is a Schottky diode instead of a junction diode.

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/9.html



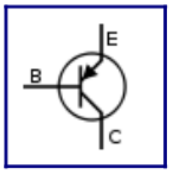
N-channel "V-MOS" transistor: (a) Cross-section, (b) schematic symbol.

The V-MOS device above takes its name from the V-shaped gate region, which increases the cross-sectional area of the source-drain path. This minimizes losses and allows switching of higher levels of power. UMOS, a variation using a U-shaped grove, is more reproducible in manufacture.

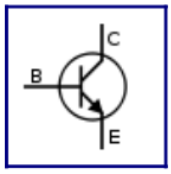
Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/10.html

Common Transistor Symbols:

BJT PNP

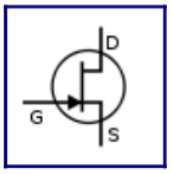


BJT NPN



P-channel JFET

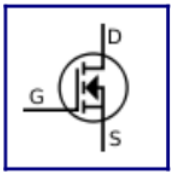
N-channel JFET



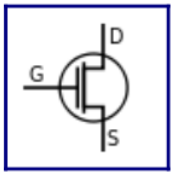
P-Channel
MOSFET enh

P-Channel
MOSFET dep

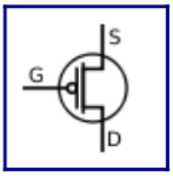
N-Channel
MOSFET dep



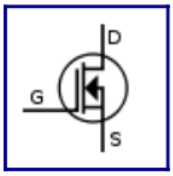
N-Channel
MOSFET enh



P-channel
MOSFET enh



N-channel
MOSFET dep

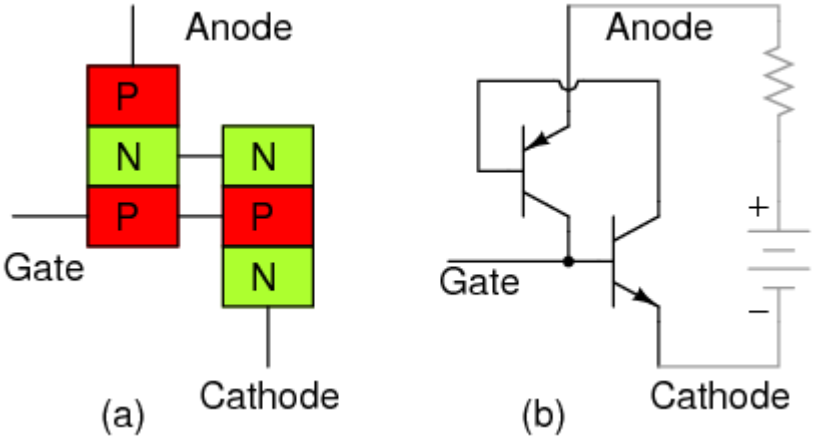


Attribution: http://en.wikipedia.org/wiki/Transistor

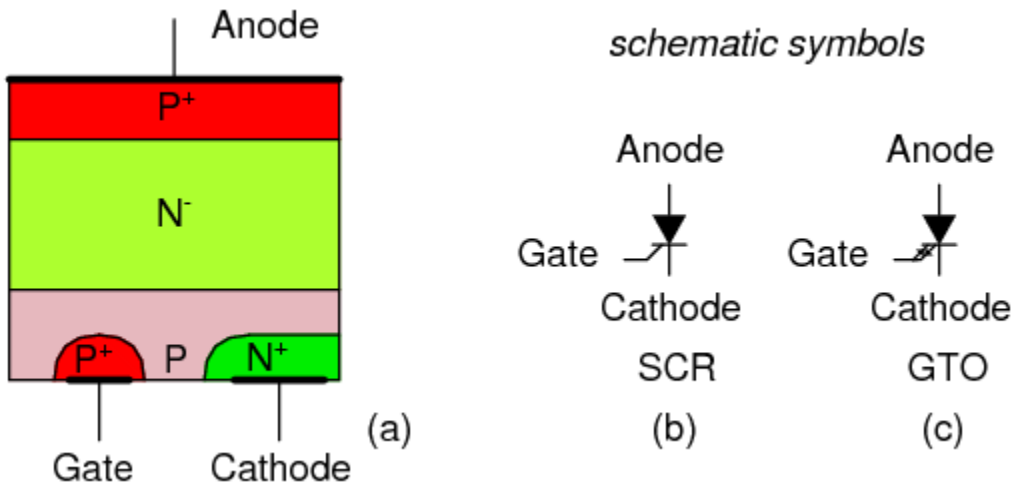
Transistor Equations

SCR & GTO Essentials

Thyristors are a broad classification of bipolar-conducting semiconductor devices *having four (or more) alternating N-P-N-P layers* and include: TRIACs, DIACs (AC diodes), gate turn off switches (GTO), unijunction transistors (UJT), silicon controlled switches (SCS), silicon controlled rectifiers (SCR) and programmable unijunction transistors (PUT).



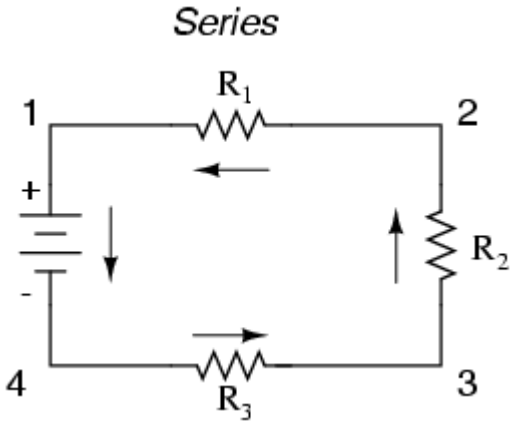
Silicon controlled rectifier (SCR): (a) doping profile, (b) BJT equivalent circuit.

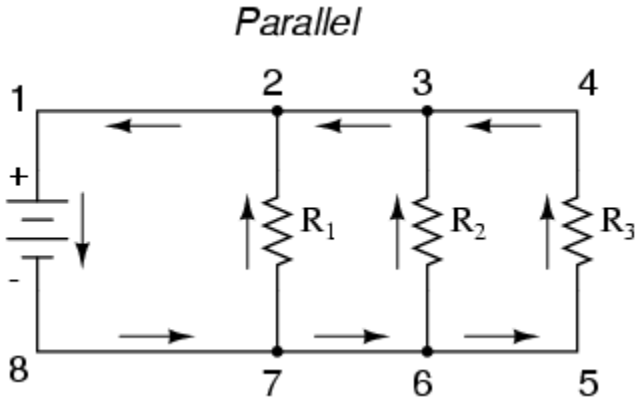


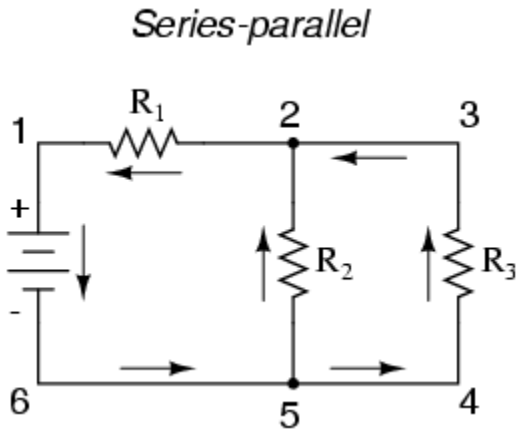
Thyristors: (a) Cross-section, (b) silicon controlled rectifier (SCR) symbol, (c) gate turn-off thyristor (GTO) symbol.

Attribution: http://www.allaboutcircuits.com/vol_3/chpt_2/11.html

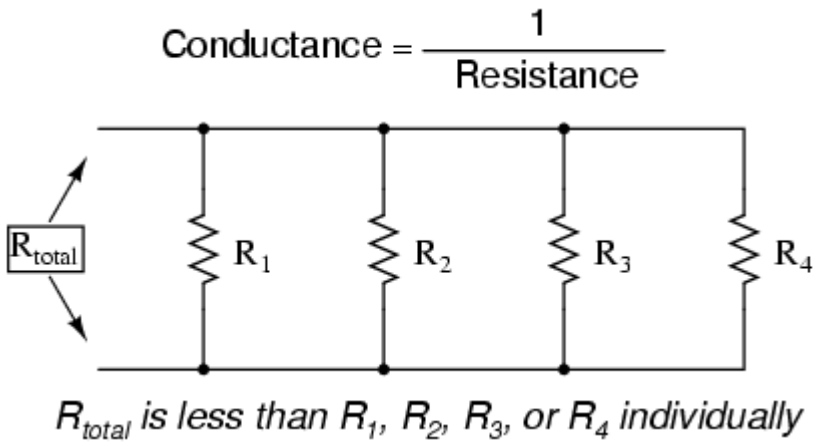
Series/Parallel Circuit Notes





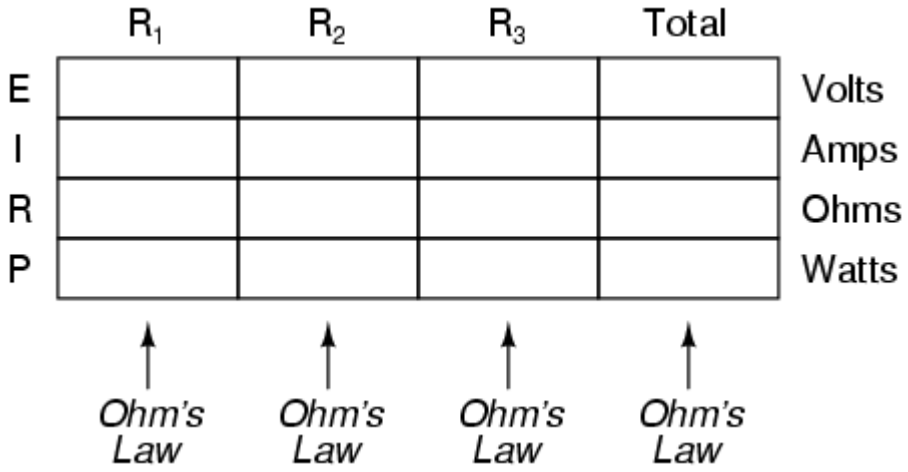


Attribution: http://www.allaboutcircuits.com/vol_1/chpt_5/1.html



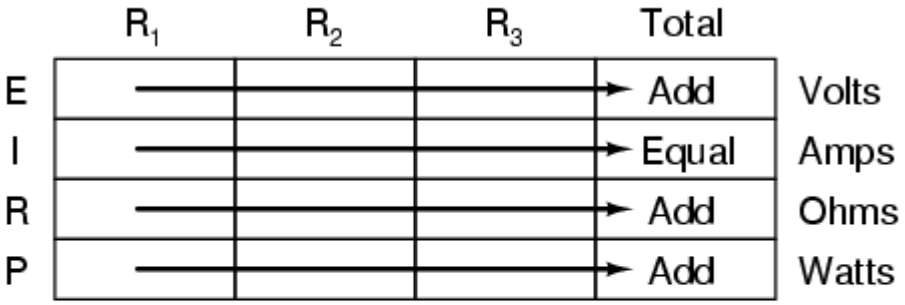
Attribution: http://www.allaboutcircuits.com/vol_1/chpt_5/4.html

The "table method" is a good way to keep the context of Ohm's Law correct for both series and parallel circuit configurations. In a table like the one shown below, you are only allowed to apply an Ohm's Law equation for the values of a single vertical column at a time:



Deriving values horizontally across columns is allowable as per the principles of series and parallel circuits:

For series circuits:



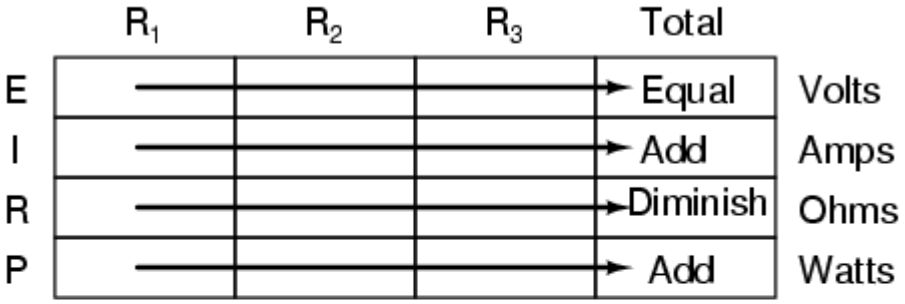
$$E_{total} = E_1 + E_2 + E_3$$

$$I_{total} = I_1 = I_2 = I_3$$

$$R_{total} = R_1 + R_2 + R_3$$

$$P_{total} = P_1 + P_2 + P_3$$

For parallel circuits:



$$E_{total} = E_1 = E_2 = E_3$$

$$I_{total} = I_1 + I_2 + I_3$$

$$R_{total} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

$$P_{total} = P_1 + P_2 + P_3$$

Not only does the "table" method simplify the management of all relevant quantities, it also facilitates cross-checking of answers by making it easy to solve for the original

unknown variables through other methods (or by working backwards to solve for the initially given values from your solutions).

For example, if you have just solved for all unknown voltages, currents, and resistances in a circuit... you can check your work by adding a row at the bottom for power calculations on each resistor, seeing whether or not all the individual power values add up to the total power.

If not, then you must have made a mistake somewhere! While this technique of "cross-checking" your work is nothing new, using the table to arrange all the data for the cross-check(s) results in a minimum of confusion.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_5/6.html

Practical Real-World-Usage Table Method Examples:
http://www.allaboutcircuits.com/vol_1/chpt_5/7.html

Voltage drop across *any* resistor $E_n = I_n R_n$

Current in a series circuit $I_{total} = \frac{E_{total}}{R_{total}}$

Voltage drop across any *series* resistor $E_n = \frac{E_{total}}{R_{total}} R_n$

. . . or . . .

$$\boxed{E_n = E_{total} \frac{R_n}{R_{total}}}$$

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_6/1.html

Current through *any* resistor $I_n = \frac{E_n}{R_n}$

Voltage in a parallel circuit $E_{total} = E_n = I_{total} R_{total}$

Current through any *parallel* resistor $I_n = \frac{I_{total} R_{total}}{R_n}$

. . . or . . .

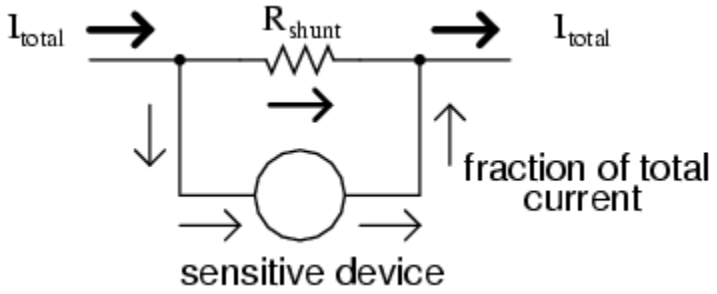
$$I_n = I_{total} \frac{R_{total}}{R_n}$$

Voltage divider formula

$$E_n = E_{total} \frac{R_n}{R_{total}}$$

Current divider formula

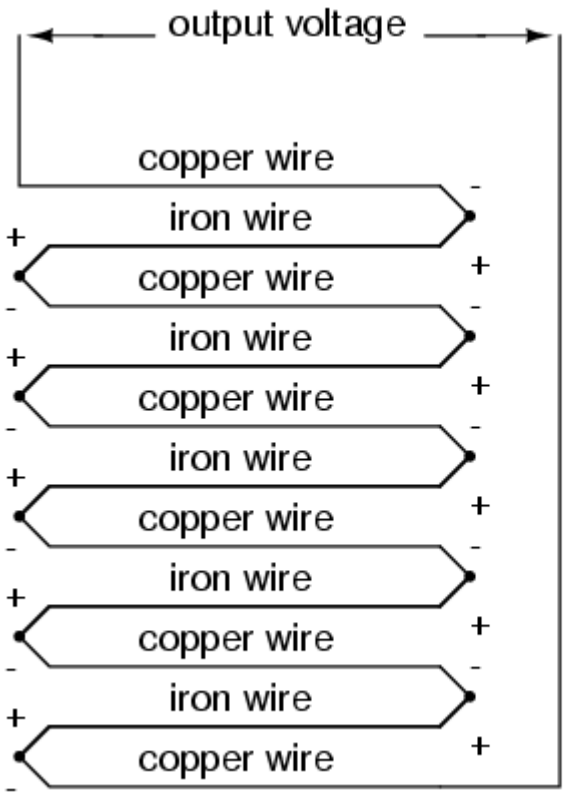
$$I_n = I_{total} \frac{R_{total}}{R_n}$$



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_6/3.html

Hot Tip: For uber-handy practical parallel/series-combination circuit *analysis techniques*: http://www.allaboutcircuits.com/vol_1/chpt_7/2.html

Thermocouple Basics

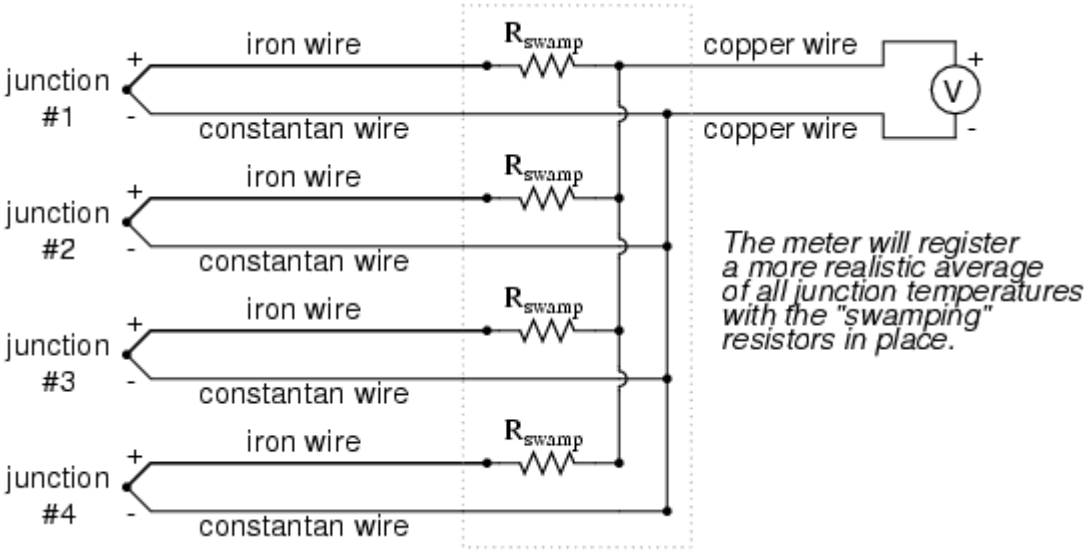


Additional resistance can be added to each of the parallel thermocouple circuit branches below to make their respective resistances more equal.

Without having to custom-size resistors for each branch, so as to make resistances precisely equal between all the thermocouples...

It is acceptable to simply install resistors with *equal values* that are **significantly higher** than the thermocouple wires' resistances; whereby the wire resistances will have a much smaller impact on the total branch resistance.

These resistors are called **swamping resistors**, because their relatively high values over-shadow or "swamp" the resistances of the thermocouple wires themselves:



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_9/5.html

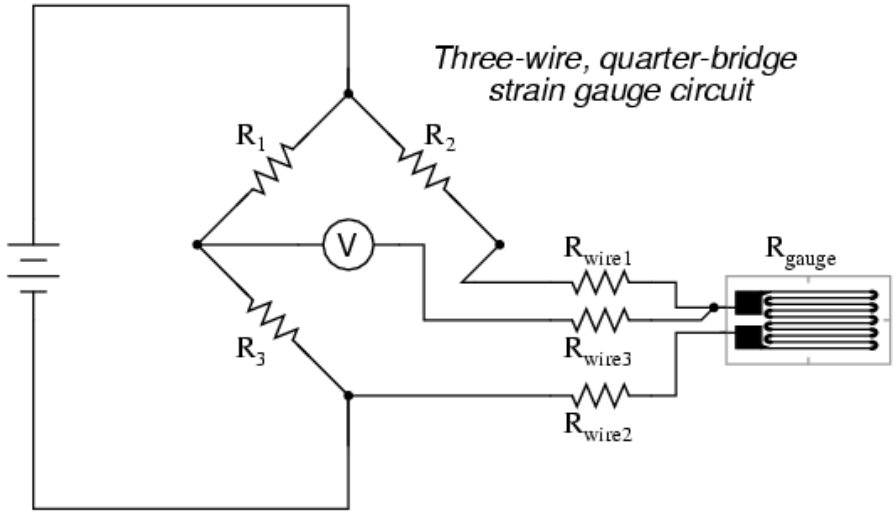
Strain Gauge Basics

If a strip of conductive metal is stretched, it will become skinnier and longer, both changes resulting in an increase of electrical resistance end-to-end. Conversely, a strip of conductive metal placed under compressive force (without buckling) will broaden and shorten.

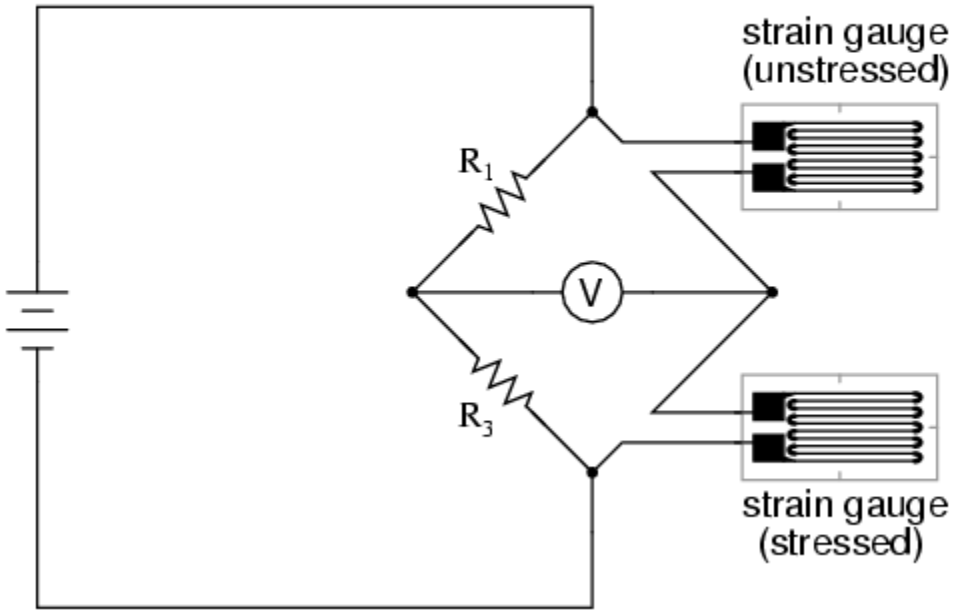
If these stresses are kept within the elastic limit of the metal strip (so that the strip does not permanently deform), the strip can be used as a measuring element for physical force; the amount of applied force inferred from measuring its resistance.

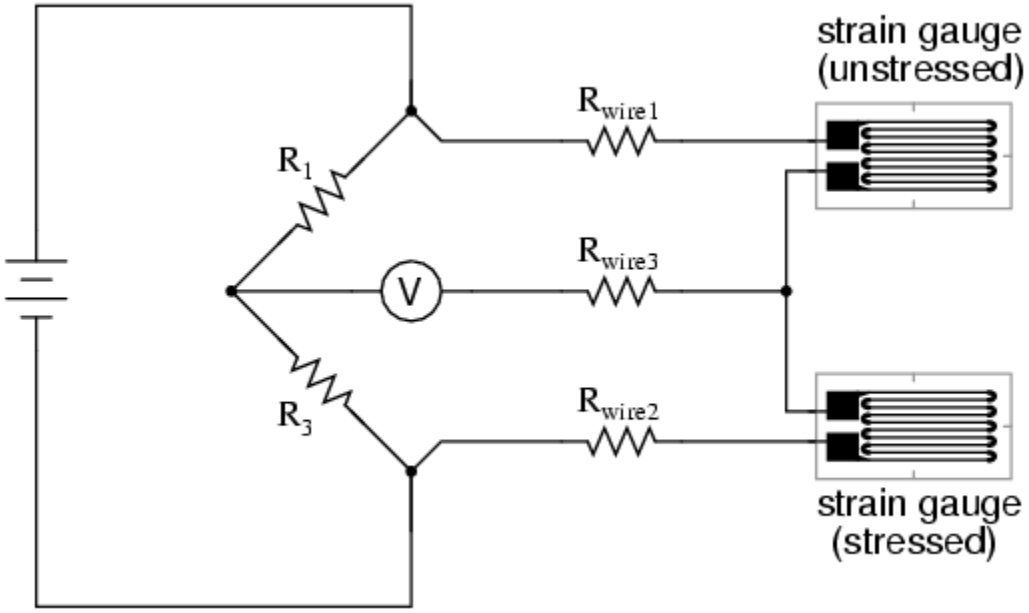
Such a device is called a strain gauge. Strain gauges are frequently used in mechanical engineering research and development to measure the stresses generated by machinery.

Note To Self: Could Muscle Wire® springs be used as the metal in a strain gauge circuit?!

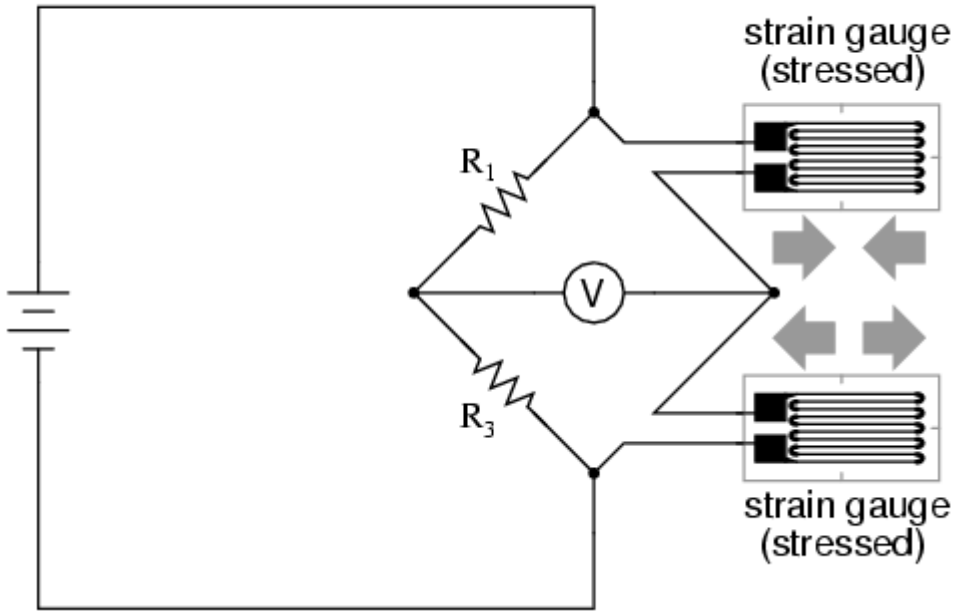


Quarter-bridge strain gauge circuit with temperature compensation

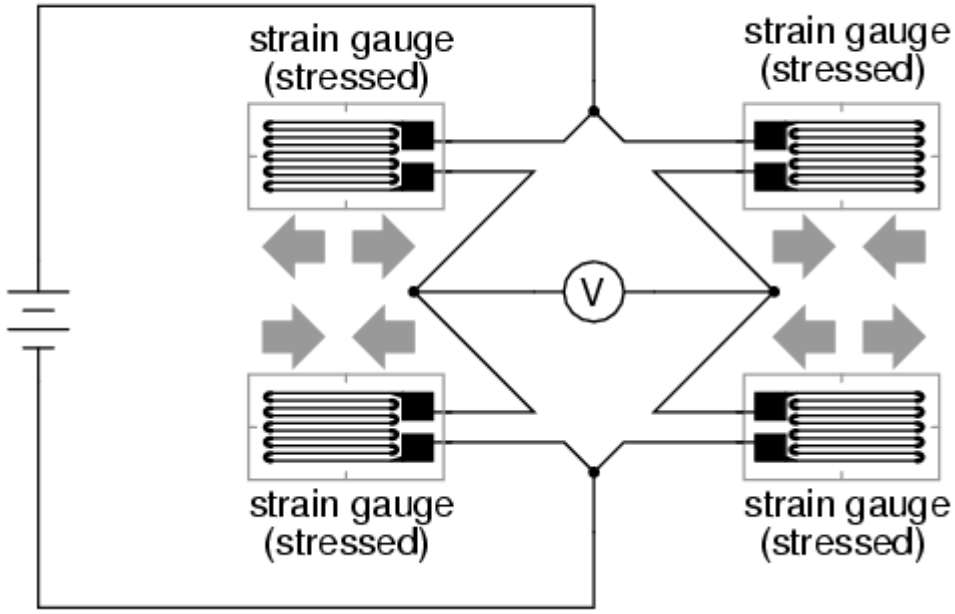


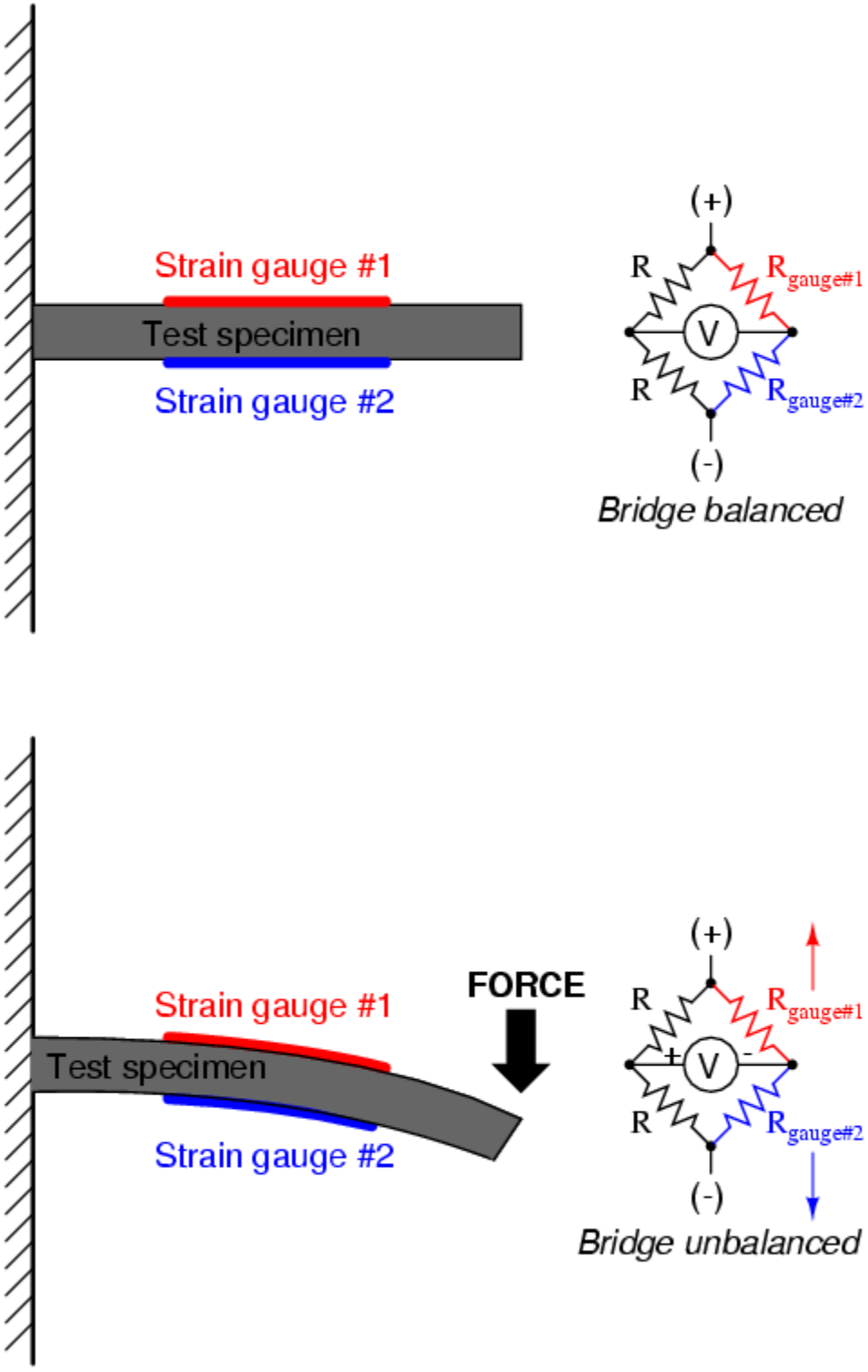


Half-bridge strain gauge circuit



Full-bridge strain gauge circuit





Attribution: http://www.allaboutcircuits.com/vol_1/chpt_9/7.html

Hot Tip: DC Network Analysts Troubleshooting Techniques

Battery Ah Capacities

The amp-hour (Ah) is a unit of battery energy capacity, equal to the amount of continuous current multiplied by the discharge time, that a battery can supply before exhausting its internal store of chemical energy.

$$\text{Continuous current (in Amps)} = \frac{\text{Amp-hour rating}}{\text{Charge/discharge time (in hours)}}$$

$$\text{Charge/discharge time (in hours)} = \frac{\text{Amp-hour rating}}{\text{Continuous current (in Amps)}}$$

An amp-hour battery rating is only an approximation of the battery's charge capacity, and should be trusted only at the current level or time specified by the manufacturer. Such a rating cannot be extrapolated for very high currents or very long times with any accuracy.

Discharged batteries lose voltage and increase in resistance. The best check for a battery is a voltage test under load.

Approximate Ah capacities of some common batteries:

- Typical automotive battery: 70 amp-hours @ 3.5 A (secondary cell)
- D-size carbon-zinc battery: 4.5 amp-hours @ 100 mA (primary cell)
- 9 volt carbon-zinc battery: 400 milliamp-hours @ 8 mA (primary cell)

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_11/3.html

Battery Array Design Considerations:

- All batteries in a series bank must have the same amp-hour rating.
- Connecting batteries in parallel increases total current capacity by decreasing total resistance; thusly also increasing overall amp-hour capacity.
- All batteries in a parallel bank must have the same voltage rating.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_11/5.html

Real-World Battery Electrical Specifications Example:

UB645 6v./4.5 Ah Sealed Lead Acid (SLA) Battery Rating Chart

ELECTRICAL SPECIFICATIONS					
Capacity	20 hour rate (0.35A) 10 hour rate (0.63A) 5 hour rate (1.12A) 1 hour rate (4.2A)	4.5 Ah 3.6 Ah 3.2 Ah 2.4 Ah	Max. Discharge Current (5 sec.) 32A		
Capacity affected by Temperature	40ºC(104ºF) 20ºC(68ºF) 0ºC(32ºF)	105% 100% 85%	Constant Voltage Charge	Cycle	Initial Charging Current less than 1.2A Voltage 7.20V ~ 7.50V at 20ºC(68ºF) Temp. Coefficient-15mV/ºC
Internal Resistance Fully charged battery (20ºC, 68ºF) 55milliohms				Standby	Initial Charging Current less than 1.2A Voltage 6.75V ~ 6.90V at 20ºC (68ºF) Temp. Coefficient-10mV/ºC

Attribution: www.ebatteriestogo.com/SpecSheets/UB645.pdf

Conductor Ampacities & Resistances

Free-Air Copper Ampacities @ 30°C

INSULATION TYPE:	RUW, T TW	THW, THWN RUH	FEP, FEPB THHN, XHHW
Size AWG	Current Rating @ 60 degrees C	Current Rating @ 75 degrees C	Current Rating @ 90 degrees C
20	*9		*12.5
18	*13		18
16	*18		24
14	25	30	35
12	30	35	40
10	40	50	55
8	60	70	80
6	80	95	105
4	105	125	140
2	140	170	190
1	165	195	220
1/0	195	230	260
2/0	225	265	300
3/0	260	310	350
4/0	300	360	405

***** = estimated values; normally, these small wire sizes are not manufactured with these insulation types

Notice the substantial ampacity differences between same-size wires with different types of insulation. This is due, again, to the thermal limits (60 o , 75 o , 90 o) of each type of insulation material.

These ampacity ratings are given for copper conductors in "free air" (maximum typical air circulation), as opposed to wires placed in conduit or wire trays.

The table fails to specify ampacities for small wire sizes, because the NEC concerns itself primarily with power wiring (large currents, big wires) vs. wires common to low-current electronic work.

The letter sequences used to identify conductor types usually refer to properties of the conductor's insulating layer(s). Some of these letters symbolize individual properties of the wire while others are simply abbreviations.

For example, the letter "T" by itself means "thermoplastic" as an insulation material, as in "TW" or "THHN." However, the three-letter combination "**MTW**" is an abbreviation for **Machine Tool Wire**, a type of wire whose insulation is made to be flexible for use in machines experiencing significant motion or vibration.

INSULATION MATERIAL
==========

C = Cotton
FEP = Fluorinated Ethylene Propylene
MI = Mineral (magnesium oxide)
PFA = Perfluoroalkoxy
R = Rubber (sometimes Neoprene)
S = Silicone "rubber"
SA = Silicone-asbestos
T = Thermoplastic
TA = Thermoplastic-asbestos
TFE = Polytetrafluoroethylene ("Teflon")
X = Cross-linked synthetic polymer
Z = Modified ethylene tetrafluoroethylene

HEAT RATING

H = 75 degrees Celsius
HH = 90 degrees Celsius

OUTER COVERING ("JACKET")

N = Nylon

SPECIAL SERVICE CONDITIONS

U = Underground
W = Wet
-2 = 90 degrees Celsius and wet

Example: "THWN" conductor = **T**hermoplastic insulation, **H**eat resistant to 75° Celsius, rated for **W**et conditions, with a **N**ylon outer jacketing.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_12/3.html

Specific Metals Resistance @ 20°C

Material	Element/Alloy	(ohm-cmil/ft)	(microohm-cm)
Nichrome	Alloy	675	112.2
Nichrome V	Alloy	650	108.1
Manganin	Alloy	290	48.21
Constantan	Alloy	272.97	45.38
Steel*	Alloy	100	16.62
Platinum	Element	63.16	10.5
Iron	Element	57.81	9.61
Nickel	Element	41.69	6.93
Zinc	Element	35.49	5.90
Molybdenum	Element	32.12	5.34
Tungsten	Element	31.76	5.28
Aluminum	Element	15.94	2.650
Gold	Element	13.32	2.214
Copper	Element	10.09	1.678
Silver	Element	9.546	1.587

***** = Steel alloy at 99.5 percent iron, 0.5 percent carbon

Notice that the figures for specific resistance in the above table are given in the very strange unit of "ohms-cmil/ft" (Ω-cmil/ft), This unit indicates what units we are expected to use in the resistance formula (R=ρl/A).

In this case, these figures for specific resistance are intended to be used when length is measured in feet and cross-sectional area is measured in circular mils.

The metric unit for *specific resistance* is the **ohm-meter (Ω-m)**, or **ohm-centimeter (Ω-cm)**, with 1.66243 x 10 -9 Ω-meters per Ω-cmil/ft (1.66243 x 10 -7 Ω-cm per Ω-cmil/ft).

In the Ω-cm column of the table, the figures are actually scaled as μΩ-cm due to their very small magnitudes. For example, iron is listed as 9.61 μΩ-cm, which could be represented as 9.61 x 10 -6 Ω-cm.

- When using the unit of Ω-meter for specific resistance in the R=ρl/A formula, the length needs to be in meters and the area in square meters.
- When using the unit of Ω-centimeter (Ω-cm) in the same formula, the length needs to be in centimeters and the area in square centimeters.

All these units for specific resistance are valid for any material (Ω-cmil/ft, Ω-m, or Ω-cm). One might prefer to use Ω-cmil/ft. However, when dealing with round wire where the cross-sectional area is already known in circular mils.

Conversely, when dealing with odd-shaped busbar or custom busbar cut out of metal stock, where only the linear dimensions of length, width, and height are known, the specific resistance units of Ω-meter or Ω-cm may be more appropriate.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_12/5.html

Resistance Formula For Conductors

Resistance values for conductors at any temperature other than the standard temperature (usually specified at 20 Celsius) on the specific resistance table must be determined via:

$$R = R_{ref}\,[1 + \alpha(T - T_{ref})]$$

Where,

R = Conductor resistance at temperature "T"

R_{ref} = Conductor resistance at reference temperature T_{ref}, usually 20° C, but sometimes 0° C.

α = Temperature coefficient of resistance for the conductor material.

T = Conductor temperature in degrees Celcius.

T_{ref} = Reference temperature that α is specified at for the conductor material.

The "alpha" (α) constant is known as the temperature coefficient of resistance, and symbolizes the resistance change factor per degree of temperature change. And just as all materials have a certain specific resistance (at 20 o C), they also change resistance according to temperature by certain amounts.

- For pure metals, this coefficient is a positive number; meaning that resistance increases with increasing temperature.
- For the elements carbon, silicon, and germanium, this coefficient is a negative number; meaning that resistance decreases with increasing temperature.
- For some metal alloys, the temperature coefficient of resistance is very close to zero; meaning that the resistance hardly changes at all with variations in temperature (a good property if you want to build a precision resistor out of metal wire!).

The following table gives the temperature coefficients of resistance for several common metals, both pure and alloy:

Temperature Coefficients of Resistance @ 20°C

Material	Element/Alloy	"alpha" per degree Celsius
Nickel	Element	0.005866
Iron	Element	0.005671
Molybdenum	Element	0.004579
Tungsten	Element	0.004403
Aluminum	Element	0.004308
Copper	Element	0.004041
Silver	Element	0.003819
Platinum	Element	0.003729
Gold	Element	0.003715
Zinc	Element	0.003847
Steel*	Alloy	0.003
Nichrome	Alloy	0.00017
Nichrome V	Alloy	0.00013
Manganin	Alloy	+/- 0.000015
Constantan	Alloy	-0.000074

***** = Steel alloy at 99.5 percent iron, 0.5 percent carbon

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_12/6.html

Capacitance Keynotes

Dielectric Strength

An insulating material's breakdown voltage is known as dielectric strength; sometimes listed in terms of volts per mil (1/1000 of an inch), or kilovolts per inch (the two units are equivalent).

However, in practice it has been found that the relationship between breakdown voltage and thickness is not exactly linear. An insulator three times as thick has a dielectric strength slightly less than 3 times as much. However, for rough estimation use, volt-per-thickness ratings are fine.

Material*	Dielectric strength (kV/inch)
Vacuum	20
Air	20 to 75
Porcelain	40 to 200
Paraffin Wax	200 to 300
Transformer Oil	400
Bakelite	300 to 550
Rubber	450 to 700
Shellac	900
Paper	1250
Teflon	1500
Glass	2000 to 3000
Mica	5000

***** = Materials listed are specially prepared for electrical use.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_12/8.html

Dielectric Material

All other factors being equal, greater permittivity of the dielectric gives greater capacitance; less permittivity of the dielectric gives less capacitance.

Explanation: Although its complicated to explain, some materials offer less opposition to field flux for a given amount of field force. Materials with a greater permittivity allow for

more field flux (offer less opposition), and thus a greater collected charge, for any given amount of field force (applied voltage).

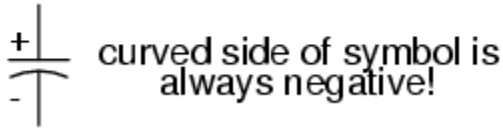
"Relative" permittivity means the permittivity of a material, relative to that of a pure vacuum. The greater the number, the greater the permittivity of the material.

Glass, for instance, with a relative permittivity of 7, has seven times the permittivity of a pure vacuum, and consequently will allow for the establishment of an electric field flux seven times stronger than that of a vacuum, all other factors being equal.

The following is a table listing the relative permittivities (also known as the "dielectric constant") of various common substances:

Material	Relative permittivity (dielectric constant)
Vacuum	1.0000
Air	1.0006
PTFE, FEP ("Teflon")	2.0
Polypropylene	2.20 to 2.28
ABS resin	2.4 to 3.2
Polystyrene	2.45 to 4.0
Waxed paper	2.5
Transformer oil	2.5 to 4
Hard Rubber	2.5 to 4.80
Wood (Oak)	3.3
Silicones	3.4 to 4.3
Bakelite	3.5 to 6.0
Quartz, fused	3.8
Wood (Maple)	4.4
Glass	4.9 to 7.5
Castor oil	5.0
Wood (Birch)	5.2
Mica, muscovite	5.0 to 8.7
Glass-bonded mica	6.3 to 9.3
Porcelain, Steatite	6.5
Alumina	8.0 to 10.0
Distilled water	80.0
Barium-strontium-titanite	7500

Electrolytic ("polarized") capacitor



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_13/5.html

An approximation of capacitance for any pair of separated conductors can be found with this formula:

$$C = \frac{\varepsilon A}{d}$$

Where,

C = Capacitance in Farads

ε = Permittivity of dielectric (absolute, not relative)

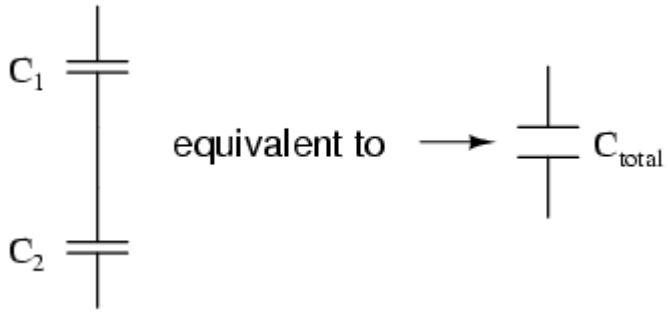
A = Area of plate overlap in square meters

d = Distance between plates in meters

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_13/3.html

When capacitors are connected in series, the total capacitance is less than any one of the series capacitors' individual capacitances. If two or more capacitors are connected in series, the overall effect is that of a single (equivalent) capacitor having the sum total of the plate spacings of the individual capacitors.

As we've just seen, an increase in plate spacing, with all other factors unchanged, results in decreased capacitance. Thus, the total capacitance is less than any one of the individual capacitors' capacitances.



The formula for calculating the series total capacitance is the same formula as for calculating parallel resistances:

$$\varepsilon = \varepsilon_0 K$$

Where,

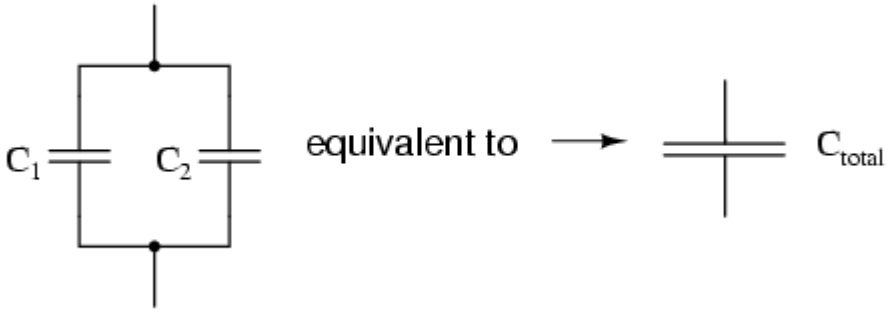
ε_0 = Permittivity of free space

$\varepsilon_0 = 8.8562 \times 10^{-12}$ F/m

K = Dielectric constant of material between plates (see table)

When capacitors are connected in parallel, the total capacitance is the sum of the individual capacitors' capacitances.

If **two or more capacitors** are connected in parallel, the overall effect is that of a single equivalent capacitor having the *sum total* of the plate areas of the individual capacitors. As we've just seen, an increase in plate area, with all other factors unchanged, results in increased capacitance.

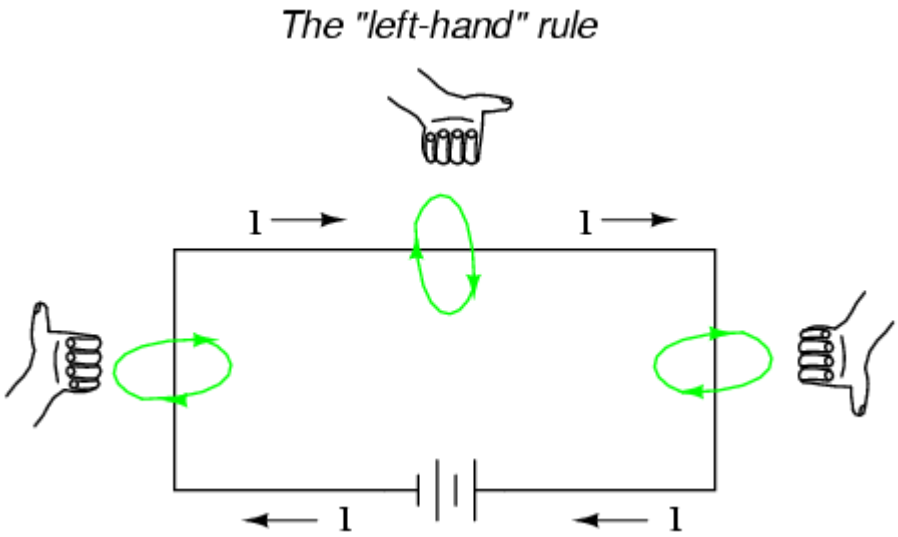


Attribution: http://www.allaboutcircuits.com/vol_1/chpt_13/4.html

Inductor Essentials

Detailed experiments showed that the magnetic field produced by an electric current is always oriented perpendicular to the direction of flow. A simple method of showing this relationship is called the **left-hand rule**.

Simply stated, the left-hand rule says that the magnetic flux lines produced by a current-carrying wire will be oriented the same direction as the curled fingers of a person's left hand (in the "hitchhiking" position), with the thumb pointing in the direction of electron flow:

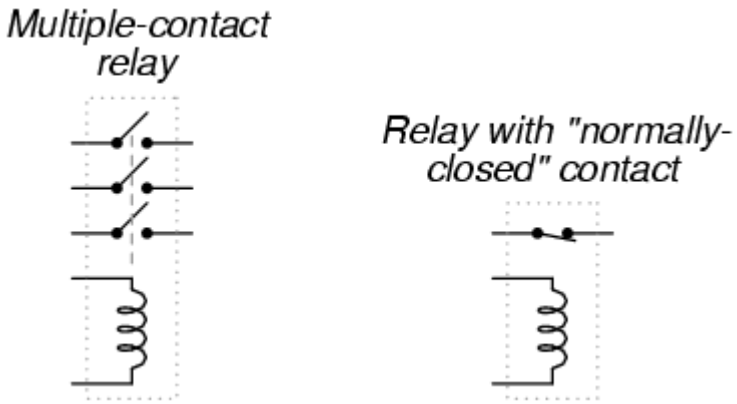


Another example is the **relay**, an electrically-controlled switch contact mechanism that can be opened and closed (actuated) by a magnetic field when an electromagnet coil is placed nearby to produce the requisite field (via a current through the coil).

In effect, this gives us a device that enables electricity to control electricity:

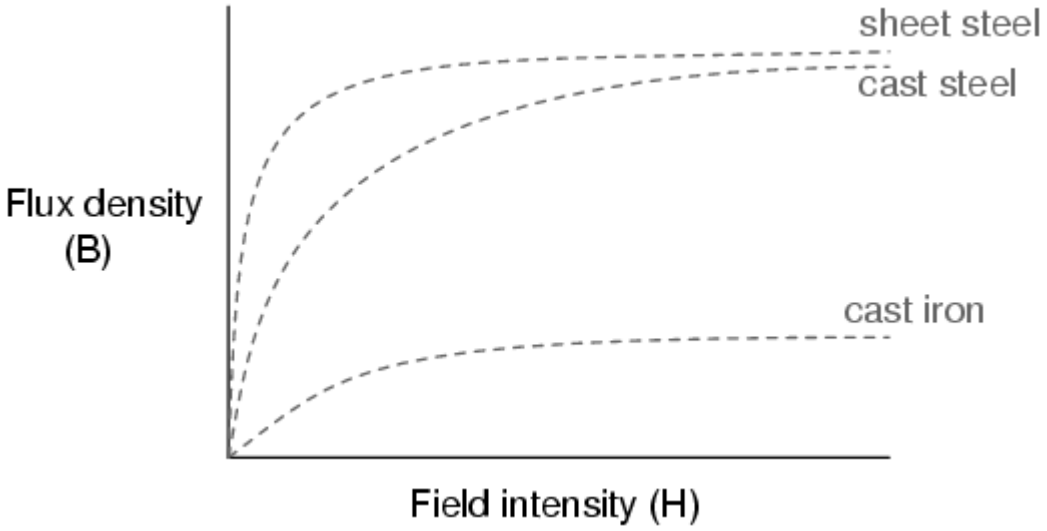
Applying current through the coil causes the switch to close.

Relays can also be constructed to actuate multiple switch contacts, or to operate in "reverse" (energizing the coil will open the switch contact, and unpowering the coil will allow it to spring closed again).



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_14/2.html

What we're trying to do here is show a mathematical relationship between field force and flux for any chunk of a particular substance, in the same spirit as describing a material's specific resistance in ohm-cmil/ft instead of its actual resistance in ohms.

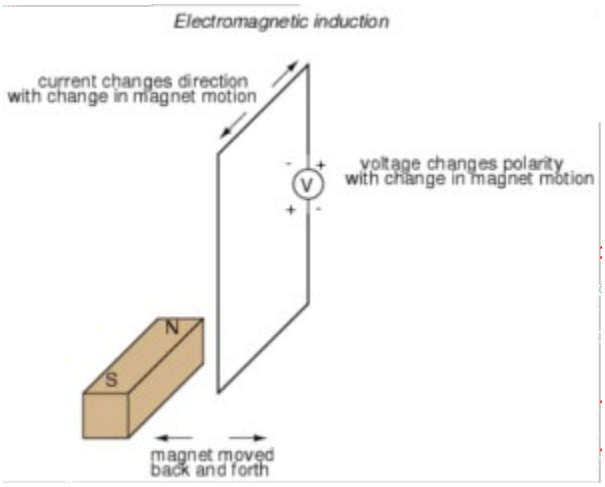


This is called the *normal magnetization curve*, or **B-H curve**, for any particular material. Notice how the flux density for any of the above materials (cast iron, cast steel, and sheet steel) levels off with increasing amounts of field intensity.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_14/4.html

An easy way to create a magnetic field of changing intensity is to move a permanent magnet next to a wire or coil of wire. Remember: the magnetic field must increase or

decrease in intensity perpendicular to the wire (so that the lines of flux "cut across" the conductor), or else no voltage will be induced:



Faraday was able to mathematically relate the rate of change of the magnetic field flux with induced voltage.

Note the use of a lower-case letter "e" for voltage; the *instantaneous voltage*, or voltage at a specific point in time, rather than a steady, stable voltage:

$$e = N\frac{d\Phi}{dt}$$

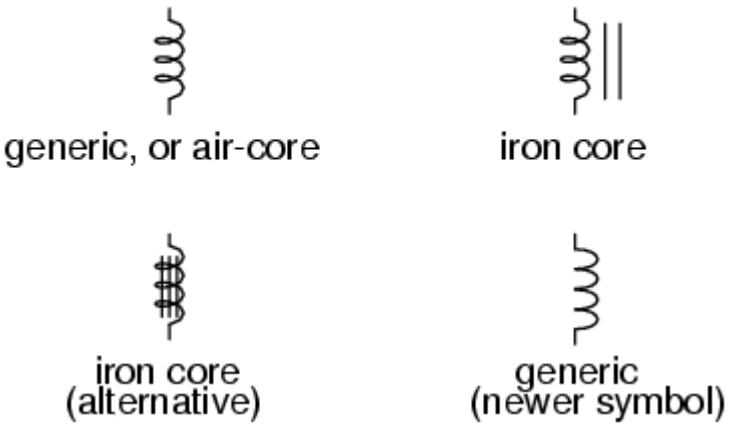
Where,

e = (Instantaneous) induced voltage in volts
N = Number of turns in wire coil (straight wire = 1)
Φ = Magnetic flux in Webers
t = Time in seconds

The "d" terms are standard calculus notation, representing rate-of-change of flux over time. "N" stands for the number of turns, or wraps, in the wire coil (assuming that the wire is formed in the shape of a coil for maximum electromagnetic efficiency).

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_14/5.html

Inductor symbols



Attribution: http://www.allaboutcircuits.com/vol_1/chpt_15/1.html

"Ohm's Law" for an inductor

$$v = L\frac{di}{dt}$$

Where,

v = Instantaneous voltage across the inductor

L = Inductance in Henrys

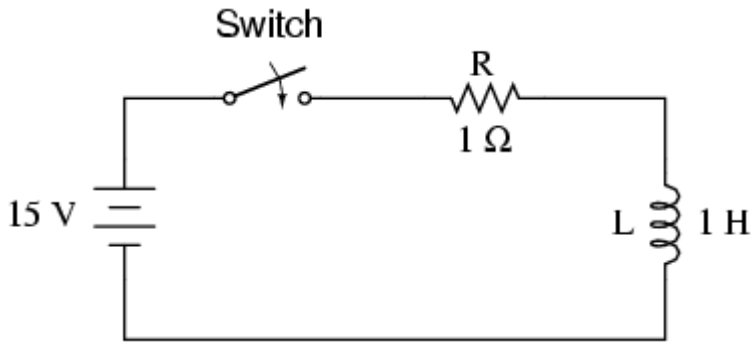
$\frac{di}{dt}$ = Instantaneous rate of current change (amps per second)

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_15/2.html

A fully discharged inductor (no magnetic field), having zero current through it, will initially act as an open-circuit when attached to a source of voltage (as it tries to maintain zero current), dropping maximum voltage across its leads.

Over time, the inductor's current rises to the maximum value allowed by the circuit, and the terminal voltage decreases correspondingly.

Once the inductor's terminal voltage has decreased to a minimum (zero for a "perfect" inductor), the current will stay at a maximum level, and it will behave essentially as a short-circuit.



Voltage across the inductor is determined by calculating how much voltage is being dropped across R, given the current through the inductor, and subtracting that voltage value from the battery to see what's left.

When the switch is first closed, the current is zero, then it increases over time until it is equal to the battery voltage divided by the series resistance of 1Ω. This behavior is *precisely opposite* of a series resistor-capacitor circuit; where current started at a maximum and capacitor voltage at zero:

Time (seconds)	Battery voltage	Inductor voltage	Current
0	15 V	15 V	0
0.5	15 V	9.098 V	5.902 A
1	15 V	5.518 V	9.482 A
2	15 V	2.030 V	12.97 A
3	15 V	0.747 V	14.25 A
4	15 V	0.275 V	14.73 A
5	15 V	0.101 V	14.90 A
6	15 V	37.181 mV	14.96 A
10	15 V	0.681 mV	14.99 A

Just as with the RC circuit, the inductor voltage's approach to 0 volts and the current's approach to 15 amps over time is **asymptotic**. For all practical purposes, though, we can say

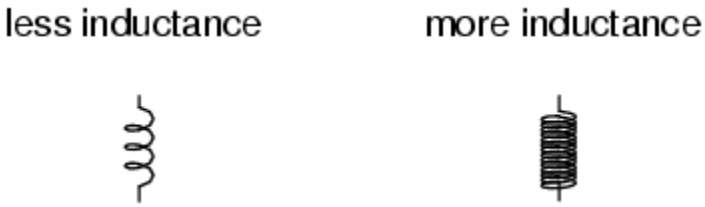
that the inductor voltage will eventually reach 0 volts and that the current will eventually equal the maximum of 15 amps.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_16/3.html

There are four basic factors of inductor construction determining the amount of inductance created. These factors all dictate inductance by affecting how much magnetic field flux will develop for a given amount of magnetic field force (i.e. current through the inductor's wire coil) as follows:

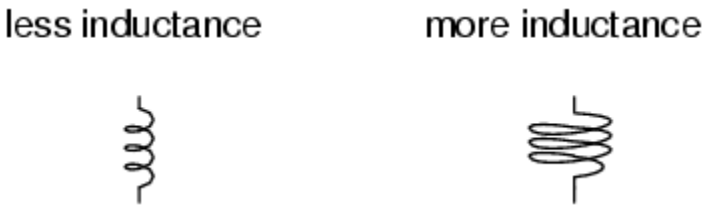
1) Number Of "Turns": All other factors being equal, a greater number of turns of wire in the coil results in greater inductance; fewer turns of wire in the coil results in less inductance.

Explanation: More turns of wire means that the coil will generate a *greater amount of magnetic field force* (measured in **amp-turns**) for a given amount of coil current:



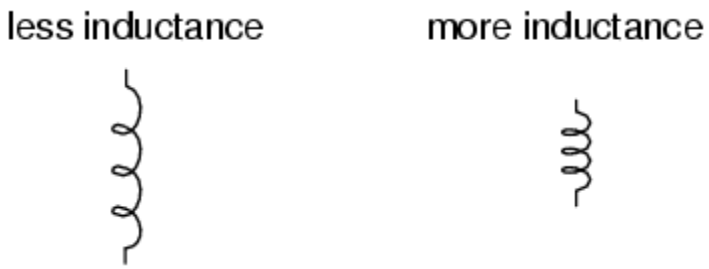
2) Coil Area: All other factors being equal, greater coil area (as measured looking lengthwise through the coil, at the cross-section of the core) results in greater inductance; less coil area results in less inductance.

Explanation: Greater coil area presents less opposition to the formation of magnetic field flux, for a given amount of field force (i.e. amp-turns).



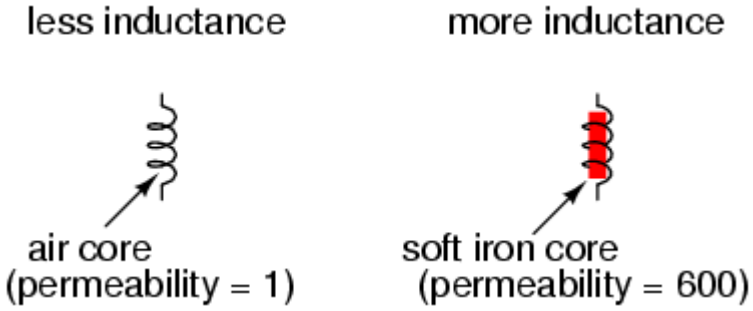
3) Coil Length: All other factors being equal, the longer the coil's length, the less inductance; the shorter the coil's length, the greater the inductance.

Explanation: A longer path for the magnetic field flux to take results in more opposition to the formation of that flux for any given amount of field force (amp-turns).



4) Core Material: All other factors being equal, the greater the magnetic permeability of the core which the coil is wrapped around, the greater the inductance; the less the permeability of the core, the less the inductance.

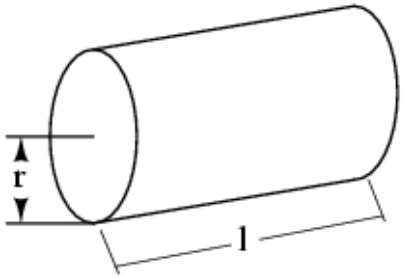
Explanation: A core material with greater magnetic permeability results in greater magnetic field flux for any given amount of field (amp-turns).



An approximation of inductance for any coil of wire can be found with this formula:

$$L = \frac{N^2 \mu A}{l}$$

$$\mu = \mu_r \mu_0$$



Where,

L = Inductance of coil in Henrys

N = Number of turns in wire coil (straight wire = 1)

μ = Permeability of core material (absolute, not relative)

μ_r = Relative permeability, dimensionless (μ_0=1 for air)

μ_0 = 1.26×10^{-6} T-m/At permeability of free space

A = Area of coil in square meters = πr^2

l = Average length of coil in meters

It must be understood that this formula yields **approximate figures only**. One reason for this is the fact that permeability changes as the field intensity varies (remember the non-linear "B/H" curves for different materials).

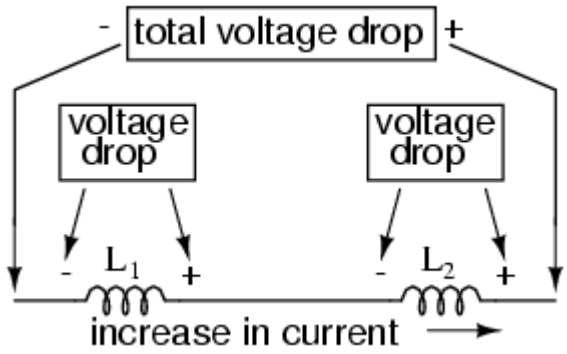
Obviously, if permeability (μ) in the equation is unstable, then the inductance (L) will also be unstable to some degree as the current through the coil changes in magnitude. If the hysteresis of the core material *is significant*, this will also have strange effects on the inductance of the coil.

Inductor designers try to minimize these effects by designing the core in such a way that its flux density never approaches saturation levels, and so the inductor operates in a more linear portion of the B/H curve.

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_15/3.html

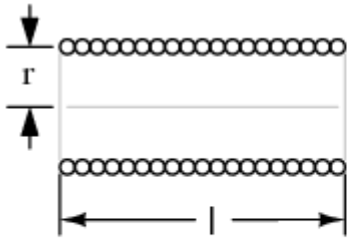
If inductors are connected together **in series** (thus sharing the same current, and seeing the same rate of change in current), then the total voltage dropped as the result of a change in current *will be additive* with each inductor.

This results in a greater total voltage than either of the individual inductors alone; greater voltage for the same rate of change in current means greater inductance.

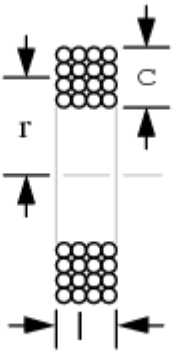


Thus, the total inductance for series inductors is *greater than any one of the individual inductors' inductances*.

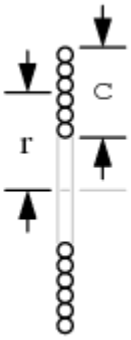
The formula for calculating the series total inductance is the same form as for calculating series resistances:



$$L = \frac{N^2 r^2}{9r + 10 \cdot l}$$



$$L = \frac{0.8 N^2 r^2}{6r + 9 \cdot l + 10c}$$



$$L = \frac{N^2 r^2}{8r + 11c}$$

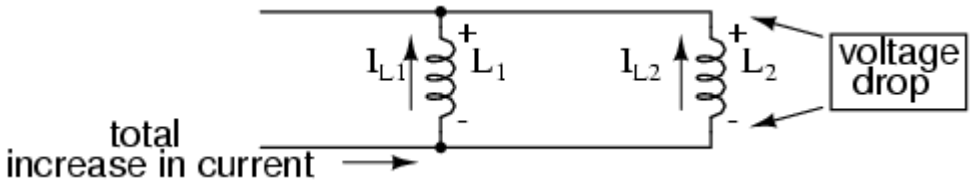
Where,

L = Inductance of coil in microhenrys
N = Number of turns of wire
r = Mean radius of coil in inches
l = Length of coil in inches
c = Thickness of coil in inches

When inductors are connected in **parallel**, the total inductance *is less than any one of the parallel inductors' inductances*. Again, remember that the definitive measure of inductance is the amount of voltage dropped across an inductor for a given rate of current change through it.

Since the current through each parallel inductor will be a fraction of the total current, and the voltage across each parallel inductor will be equal, a change in total current will result in less voltage dropped across the parallel array than for any one of the inductors considered separately.

In other words, there will be less voltage dropped across parallel inductors for a given rate of change in current than for any of those inductors considered separately, because total current divides among parallel branches. Less voltage for the same rate of change in current means less inductance.



Thus, the total inductance is less than any one of the individual inductors' inductances. The formula for calculating the parallel total inductance is the same form as for calculating parallel resistances:

AWG gauge	turns/ inch	AWG gauge	turns/ inch	AWG gauge	turns/ inch	AWG gauge	turns/ inch
10	9.6	20	29.4	30	90.5	40	282
11	10.7	21	33.1	31	101	41	32?/
12	12.0	22	3?.0	32	113	42	3?/8
13	13.5	23	41.3	33	12?/	43	421
14	15.0	24	46.3	34	143	44	4?/1
15	16.8	25	51.?	35	158	45	523
16	18.9	26	58.0	36	1?/5	46	581
1?/	21.2	2?/	64.9	3?/	198		
18	23.6	28	?/2.?	38	224		
19	26.4	29	81.6	39	248		

Attribution: http://www.allaboutcircuits.com/vol_1/chpt_15/4.html

Calculating The Time Constant Of A Circuit

The time constant of the circuit (the amount of time it takes for voltage or current values to change approximately 63 percent (from their starting values to their final values) in a transient situation.

- **Series R/C Circuit:** The time constant is equal to the total resistance in ohms multiplied by the total capacitance in farads.
- **Series L/R Circuit:** The total inductance in henrys divided by the total resistance in ohms. In either case, the time constant is expressed in units of seconds and symbolized by the Greek letter "tau" (τ).

For resistor-capacitor circuits:

$$\tau = RC$$

For resistor-inductor circuits:

$$\tau = \frac{L}{R}$$

A more universal formula for the determination of voltage and current values in transient circuits:

Universal Time Constant Formula

$$\text{Change} = (\text{Final-Start})\left(1 - \frac{1}{e^{t/\tau}}\right)$$

Where,

Final = Value of calculated variable after infinite time (its *ultimate* value)

Start = Initial value of calculated variable

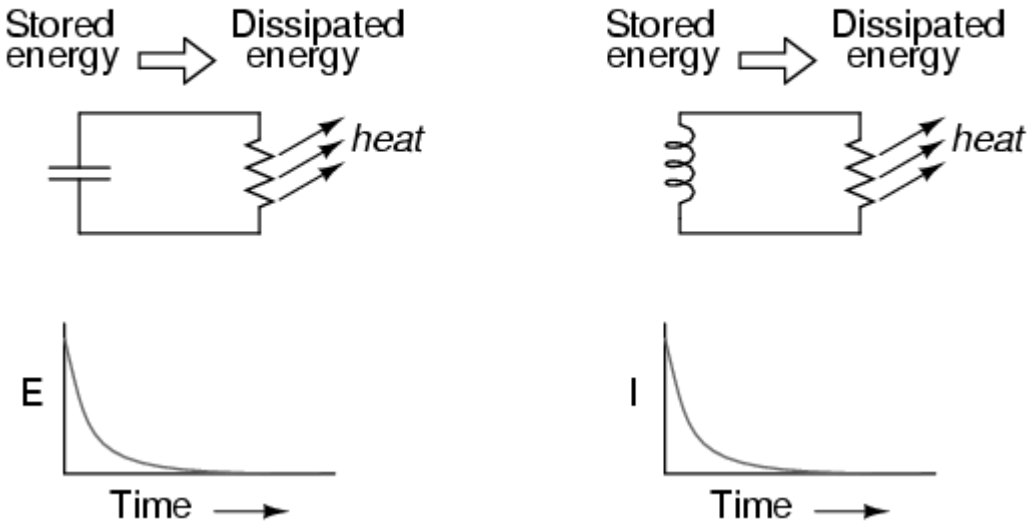
e = Euler's number (≈2.7182818)

t = Time in seconds

τ = Time constant for circuit in seconds

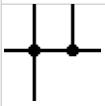
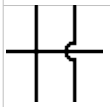
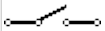
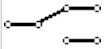
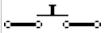
Attribution: http://www.allaboutcircuits.com/vol_1/chpt_16/4.html

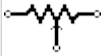
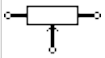
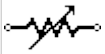
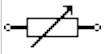
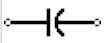
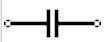
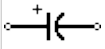
Capacitor and inductor discharge

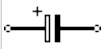
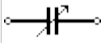
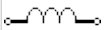
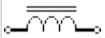
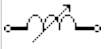
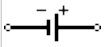
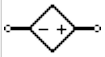


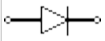
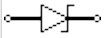
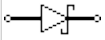
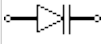
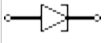
Attribution: http://www.allaboutcircuits.com/vol_1/chpt_16/5.html

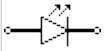
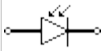
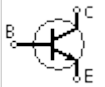
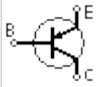
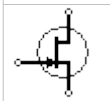
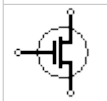
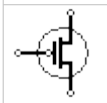
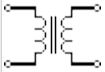
Table of Electrical Symbols

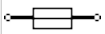
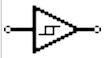
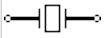
Symbol	Component name	Meaning
Wire Symbols		
	Electrical Wire	Conductor of electrical current
	Connected Wires	Connected crossing
	Not connected Wires	Wires are not connected
Switch Symbols and Relay Symbols		
	SPST Toggle Switch	Disconnects current when open
	SPDT Toggle Switch	Selects between two connections
	Pushbutton Switch (N.O)	Momentary switch - normally open
	Pushbutton Switch (N.C)	Momentary switch - normally closed
	DIP Switch	DIP switch is used for onboard configuration
	SPST Relay	Relay open / close connection by an electromagnet
	SPDT Relay	
	Jumper	Close connection by jumper insertion on pins.
	Solder Bridge	Solder to close connection

Ground Symbols		
	Earth Ground	Used for zero potential reference and electrical shock protection.
	Chassis Ground	Connected to the chassis of the circuit
	Digital / Common Ground	
Resistor Symbols		
	Resistor (IEEE)	Resistor reduces the current flow.
	Resistor (IEC)	
	Potentiometer (IEEE)	Adjustable resistor - has 3 terminals.
	Potentiometer (IEC)	
	Variable Resistor / Rheostat (IEEE)	Adjustable resistor - has 2 terminals.
	Variable Resistor / Rheostat (IEC)	
Capacitor Symbols		
	Capacitor	Capacitor is used to store electric charge. It acts as short circuit with AC and open circuit with DC.
	Capacitor	
	Polarized Capacitor	Electrolytic capacitor

	Polarized Capacitor	Electrolytic capacitor
	Variable Capacitor	Adjustable capacitance
Inductor / Coil Symbols		
	Inductor	Coil / solenoid that generates magnetic field
	Iron Core Inductor	Includes iron
	Variable Inductor	
Power Supply Symbols		
	Voltage Source	Generates constant voltage
	Current Source	Generates constant current.
	AC Voltage Source	AC voltage source
	Generator	Electrical voltage is generated by mechanical rotation of the generator
	Battery Cell	Generates constant voltage
	Battery	Generates constant voltage
	Controlled Voltage Source	Generates voltage as a function of voltage or current of other circuit element.
	Controlled Current Source	Generates current as a function of voltage or current of other circuit element.
Meter Symbols		

	Voltmeter	Measures voltage. Has very high resistance. Connected in parallel.
	Ammeter	Measures electric current. Has near zero resistance. Connected serially.
	Ohmmeter	Measures resistance
	Wattmeter	Measures electric power
Lamp / Light Bulb Symbols		
	Lamp / light bulb	Generates light when current flows through
	Lamp / light bulb	
	Lamp / light bulb	
Diode / LED Symbols		
	Diode	Diode allows current flow in one direction only (left to right).
	Zener Diode	Allows current flow in one direction, but also can flow in the reverse direction when above breakdown voltage
	Schottky Diode	Schottky diode is a diode with low voltage drop
	Varactor / Varicap Diode	Variable capacitance diode
	Tunnel Diode	

	Light Emitting Diode (LED)	LED emits light when current flows through
	Photodiode	Photodiode allows current flow when exposed to light
Transistor Symbols		
	NPN Bipolar Transistor	Allows current flow when high potential at base (middle)
	PNP Bipolar Transistor	Allows current flow when low potential at base (middle)
	Darlington Transistor	Made from 2 bipolar transistors. Has total gain of the product of each gain.
	JFET-N Transistor	N-channel field effect transistor
	JFET-P Transistor	P-channel field effect transistor
	NMOS Transistor	N-channel MOSFET transistor
	PMOS Transistor	P-channel MOSFET transistor
Misc. Symbols		
	Motor	Electric motor
	Transformer	Change AC voltage from high to low or low to high.
	Electric bell	Rings when activated
	Buzzer	Produce buzzing sound

	Fuse	The fuse disconnects when current above threshold. Used to protect circuit from high currents.
	Fuse	
	Bus	Contains several wires. Usually for data / address.
	Bus	
	Bus	
	Optocoupler / Opto-isolator	Optocoupler isolates onnection to other board
	Loudspeaker	Converts electrical signal to sound waves
	Microphone	Converts sound waves to electrical signal
	Operational Amplifier	Amplify input signal
	Schmitt Trigger	Operates with hysteresis to reduce noise.
	Analog-to-digital converter (ADC)	Converts analog signal to digital numbers
	Digital-to-Analog converter (DAC)	Converts digital numbers to analog signal
	Crystal Oscillator	Used to generate precise frequency clock signal
Antenna Symbols		
	Antenna / aerial	Transmits & receives radio waves

	Antenna / aerial	
	Dipole Antenna	Two wires simple antenna
Logic Gates Symbols		
	NOT Gate (Inverter)	Outputs 1 when input is 0
	AND Gate	Outputs 1 when both inputs are 1.
	NAND Gate	Outputs 0 when both inputs are 1. (NOT + AND)
	OR Gate	Outputs 1 when any input is 1.
	NOR Gate	Outputs 0 when any input is 1. (NOT + OR)
	XOR Gate	Outputs 1 when inputs are different. (Exclusive OR)
	D Flip-Flop	Stores one bit of data
	Multiplexer / Mux 2 to 1	Connects the output to selected input line.
	Multiplexer / Mux 4 to 1	
	Demultiplexer / Demux 1 to 4	Connects selected output to the input line.

Attribution: http://www.rapidtables.com/electric/electrical_symbols.htm

Section II: Exploring The Pleasures Of Neurohacking

Pt. 1: Brainwave Entrainment Basics

Brief Introduction

Everything you experience is both affected by and affects your brainwaves; hence a distinct relationship between your brain frequencies and your perceptual life experiences dynamically shifts moment by moment.

Said another way:

- Anything that radically changes your perception radically changes your brainwave rhythms; whether it be psycho-meds, recreational drugs, sex, meditation, solving puzzles, etc.

- Conversely, anything that radically alters your brainwaves, radically alters your mood and perception.

Therefore, understanding *optimal brainwave frequency profiles* is the first step to unleash the true power of your mind to do amazing Ψ stuff, rapidly heal, consciously manifest your desires, etc.

For example, your success with relaxing, focusing, and getting into the Zone will expedite your subconscious mind accepting your visual and verbal suggestions; whereby effectively changing its programming (i.e. removing all desire for burdensome additions, speeding up the healing process, instilling desirable habits, speed-learning new skills, etc.)

With a combination of the right tools, knowledge, and some practice... you too can quickly get your brain waves into the Zone **at will**. Then it's simply a matter of implanting the new thoughts and behaviors you want via congruent visual and auditory suggestions.

As luck would have it... you've definitely come to the right place; so without any further ado, let's get to it!

Brainwave Entrainment Simplified

So what exactly is “brain entrainment”?!

From BrainWorksNeurotherapy.com:

"Brainwave entrainment is a method to stimulate the brain into entering a specific state by using a pulsing sound, light, or electromagnetic field. The pulses elicit the brain's 'frequency following' response, encouraging the brainwaves to align to the frequency of a given beat."

Simply put, brainwave entrainment (aka "**hemi-sync**") is a type of resonance ; whereby two unsynchronized brainwave rhythms will synchronize with each other in a relatively short amount of time.

You're likely familiar with the proverbial hypnotist mesmerizing their subject with a pocket watch swinging back and forth like a pendulum. Interestingly, it actually has historical roots.

Back in the 1656, Christian Huygens discovered that two pendulums started at different times (completely out of sync) would synchronize, swinging in opposite directions; hence he ended up stumbling upon the realization that brainwave frequencies can also resonate with an external stimulus.

Here's a really short, yet strangely satisfying video demonstrating this pendulum effect.

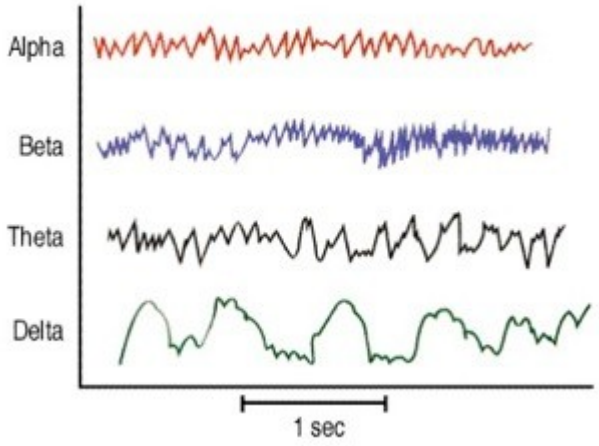


Image via BinauralBrains.com

- **Beta stimulation**: Beta wave stimulation can improve attention, relieve short-term stress, decrease headaches, and decrease behavioral problems and emotion-based exhaustion, amongst other benefits.
- **Alpha stimulation**: Alpha wave stimulation benefits include relaxation, stress reduction, pain relief, and improved recognition, to name just a few.
- **Theta stimulation**: Theta wave stimulation is also great for deep relaxation, as well as exploring lucid dreaming, altered states of consciousness, and Ψ experimentation, for example.

- **Delta stimulation**: Delta wave stimulation can improve short-term stress, as well as relief from headaches/migraines, etc.

Binaural Beats vs. Isochronic Tones

Both binaural beats and isochronic tones can initiate hemi-sync; and neither is necessarily 'better' or 'worse' than the other. Specific case-use and how each gets the job done does varies greatly, however.

Binaural Beats: Binaural beats work by broadcasting a different frequency into each ear.

The difference between the two produces a third **beat frequency**. For example, playing a 124Hz frequency in one ear and 114Hz in the other produces a 10Hz internal beat frequency:

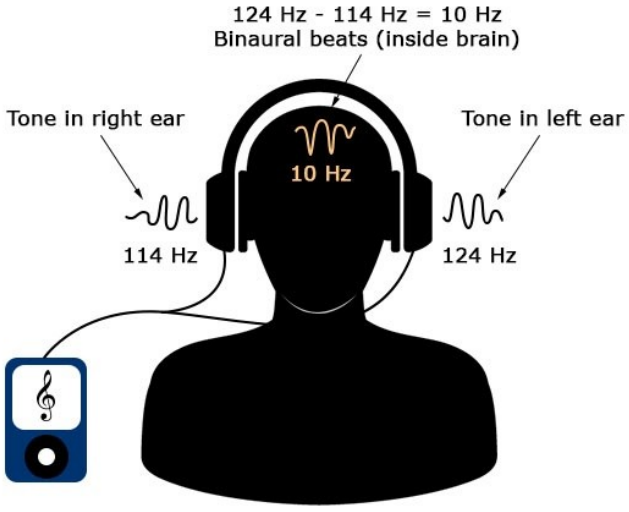


Image via BeBrainFit.com

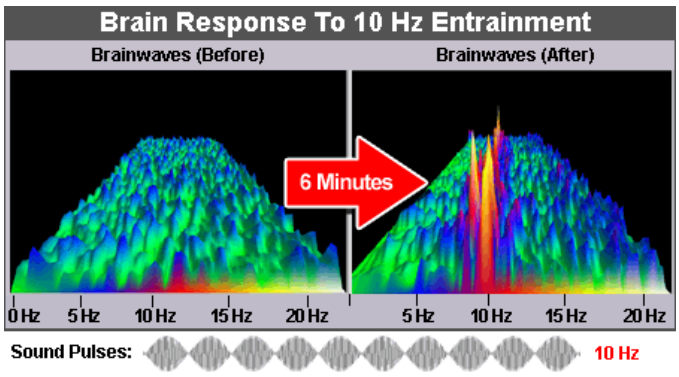


Image via TransparentcCorp.com

Math Geek Alert: https://en.wikipedia.org/wiki/Beat_(acoustics)#Binaural_beats

Unlike isochronic tones, binaural beats have a *continuous wave pattern* and therefore tend to blend better with ambient music (making them the popular choice for hypnosis CD recordings and the like).

When listening to binaural beats, stereo headphones are **highly desirable** for maximum benefit in the shortest amount of time. However, centering yourself between *two equi-distantly spaced stereo speakers* (relative to your ears) will also work; although it can take awhile longer to get the equivalent results, as compared to if you were wearing head-phones.

Likewise, closing your eyes is not absolutely necessary, albeit it also greatly helps you to gain maximum benefit in the shortest amount of time (considerably less distractions for the brain to deal with).

Isochronic Tones: Isochronic tones are monaural beats comprised of *evenly spaced pulses* that combine with a carrier frequency to make a single tone, and are designed to evoke a **frequency-following response**; whereby able to harmonize with the brain efficiently.

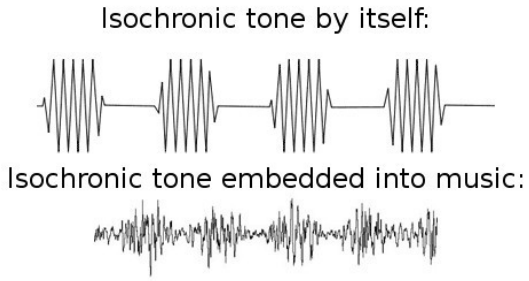


Image via MindAmend.com

Supposedly, these spaces make isochronic tones more efficient and effective than binaural beats, albeit they do initially take a bit of getting used to.

However, isochronic tones are *not dependent* upon headphones or spacing your speakers equidistantly from your ears to achieve the desired result. The important thing is that you are able to hear the tone.

In Short:

- Binaural beats are best for *submersive experimentation* (i.e. eyes closed, lights dimmed, and headphones on).

- Isochronic tones are definitely more convenient, and therefore ideally suited to set-and-forget sessions (i.e. eyes wide open, going about your business as usual).

And that's the *true beauty* of brainwave entrainment - it doesn't require any "effort" to use, once you've selected the right tone/pulse profile that suits your needs at the moment.

Simply listen and let go with binaural beats, for a more "submersive" experience or go about your business as usual if listening to isochronic tones.

While still on the topic of isochronic tones, I highly encourage you to play around with the following two free online tools to gain a deeper understanding:

- https://mynoise.net/NoiseMachines/isochronicBrainwaveGenerator.php
- https://brainaural.com/

Also, here's a few more brainwave entrainment/relaxation toys to play with:

- Audio Check (Great little toolbox of goodies for calibrating your Arduino brainhacking projects, frequency measuring equipment, etc.)
- Chameleon (User-definable relaxing music generator)
- White Noise & Co. (User-definable white noise generator)
- Jungle Life (User-definable jungle noise generator)
- Rain Noise (User-definable rain noise generator)
- Tibetan Choir (User-definable overtone throat singing generator)

There's actually quite a few more here, but these are ones I wanted quick and easy access to. On a related side note, **light-based brainwave entrainment** (i.e. pulsating "stereo" LEDs) is the light-based version of binaural beats.

Brainhacker Trivia: Binaural beats were first discovered by Heinrich Wilhelm Dove in 1839

Scientific Benefits Of Brainwave Entrainment Technology

Published scientific findings concerning brainwave entrainment largely suggest that it may be a successful adjunct treatment when used with the *proper frequency profile*.

Quite frankly, this area of study is still very much in its infancy; nonetheless, here's a few interesting scientific insights I've come across:

- https://www.transparentcorp.com/downloads/Huang-PsychologicalEffectsBrainwaveEntrainment.pdf
- http://www.trainyourbraintransformyourlife.com/Alert%20clinical%20study.pdf

- http://www.researchgate.net/publication/247871689_Photic_Driving_and_Altered_States_of_Consciousness_An_Exploratory_Study
- http://www.ncbi.nlm.nih.gov/pubmed/6945531
- http://www.ncbi.nlm.nih.gov/pubmed/2708044
- http://onlinelibrary.wiley.com/doi/10.1111/j.1526-4610.1985.hed2508444.x/abstract
- http://www.scirp.org/journal/PaperInformation.aspx?paperID=36722#.VRsZxPnF9Dw
- http://www.ncbi.nlm.nih.gov/pubmed/16706812
- http://www.ncbi.nlm.nih.gov/pubmed/6945531
- http://www.ncbi.nlm.nih.gov/pubmedhealth/PMH0027030/
- http://www.ncbi.nlm.nih.gov/pubmed/18780583
- http://www.ncbi.nlm.nih.gov/pubmed/23862643

Brainhacker Trivia: In 1997, a Japanese television station broadcast a light and sound strobe-like effect embedded into a cartoon. Though lasting only a mere 5 seconds, 700 children ended up in hospital with seizures, and untold thousands more experienced headaches, dizziness, and nausea.

"With brain wave entrainment technology, changing brain wave states is an instantaneous and effortless process. The 'periodic stimulus' can be sound, vibrations and/or light.

We have found that we get the best results with blinking lights which are experienced through closed eyelids. This is only problematic for people with existing diagnosis of photo-induced epilepsy, as blinking lights can induce a seizure in them."

- David Mager

Editor Note: In regard to the above quote, light-based brain entrainment is definitely quicker (and often times more intense) than sound-based brain entrainment *on a one-to-one basis*.

However, if you want to *take it to the next level*, be sure to read the entire "**Cossum's Awesome Enhanced Brain Machine**" section. By properly combining both types of entrainment, truly powerful stuff happens.

Part 2: Single Individual Frequency Effects

Reputed Effects Of Single Individual Brainwave Frequencies

Attribution: The following list of frequencies in Hertz (Hz) is a massively stripped down and streamlined portion of the Brainwave/Cymatic Frequencies; originally compiled by Michael Triggs, and further enhanced by Mark Maffei via CC BY SA *provided this entire attribution remains fully intact*. However, you can slightly reword it to fit the flow of your remix.

- 0.5 - very relaxing against headache and lower back pain; Thyroid, reproductive, excretory stimulant, whole brain toner.
- 0.5-1.5 Pain relief; endorphins, better hypnosis
- 0.5-4 Trance, suspended animation, anti-aging*****. Reduces amount of cortisol, a hormone associated with stress & aging. Increases the levels of DHEA (anti-aging) & melantonin (decreases aging process).

In conjunction with other frequencies in a waking state: "Delta acts as a form of radar – seeking out information – reaching out to understand on the deepest unconscious level things that we can't understand through [conscious] thought process."

Conducive to miracle type healing, divine knowledge, inner being and personal growth, rebirth, trauma recovery; "one with the universe" experiences, near death experience; characterized by "unknowing", merely a blissful "being" state such as deep sleep or coma.

*****The anti-aging info comes from a Brainwave Generator preset authored by TheMind2 - he uses binaurals at 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5 & 4.0 HZ. They all play simultaneously.

- 0.9 - Euphoria
- 0.95 - Whiplash
- 1.0 - Feeling of well-being, pituitary stimulation to release growth hormone; overall view of inter-relationships; harmony & balance.
- 1.05 - Helps hair grow + get its color back [RA]; pituitary stimulation to release growth hormone (helps develop muscle, recover from injuries, rejuvenation effects).
- 1.2 - Used on headaches.
- 1.45 - Tri-thalamic entrainment format. According to Ronald deStrulle, creates entrainment between hypothalamus, pituitary & pineal. May benefit dyslexics + people with Alzheimer's.

- 1.5 - Abrahams Universal Healing Rate; sleep; individuals whose ailments have manifested into the fourth stage of Chronic Fatigue, where some form of disease is apparent, experienced a release from the negative sensation of their symptoms when moved into 1.5Hz.
- 1.8 - Sinus Congestion seems to clear centering around 1.8 Hz (tested with binaural beats, primarily).
- 2.0 - Nerve regeneration.
- 2.06 - Associated with coccyx (small triangular bone at end of the spinal column).
- 2.30 - Associated with genitals.
- 2.5 - Pain relief, relaxation; production of endogenous opiates; use for sedative effect; reported use on bleeding, bruises, insomnia, and sinusitis. Sexual stimulation?
- 2.57 - Associated with bladder.
- 2.67 - Associated with intestines.
- 3.07 - Associated with the pelvis.
- 3.4 - Sound sleep.
- 3.44 - Associated with ovaries.
- 3.5 - Feeling of unity with everything; accelerated language retention; enhancement of receptivity; reduction in depression & anxiety; whole-being regeneration; DNA stimulation.
- 3.6 - Reduction in anger & irritability.
- 3.84 Associated with the liver/pancreas.
- 3.9 - Reduction in unsociable behavior; crystal clear meditation, lucid dreams, enhanced inner awareness, "facilitates easy access to inner resources & creates space for inner peace + self-renewal"; 1 octave below *traditional* Schumann Frequency (Earth resonance) @ ~7.8 Hz.
- 4.0 - Psi (**Ψ**) frequency “sweet spot"; seduction mindset; vital for memory & learning; subconscious problem solving/full memory scanning (if one can manage to stay awake)."Those who suffer from Chronic Fatigue exhaust very easily. When moved to 4HZ these individuals showed marked improvement in the length of time between the occurrence of exhaustion after certain exercises were completed."

FYI: According to Cavanagh (1972), theta at 4Hz corresponds to a *full memory search*. Theta, then, like alpha, is a scanning frequency. Cavanagh began by compiling a number of studies dealing with different classes of stimuli (digits, colors, letters, words, geometrical shapes, random forms, and nonsense symbols).

Each class of stimuli was found to have a characteristic reaction time. However, he found a constant of 243.2 ms when multiplying the reaction time for a single item by the maximum number of items in a given class. This indicated that each item class was scanned at a different speed, but that scanning of the full memory is always executed at a speed of 4Hz.

According to Webber (2001), after 10 minutes of listening to 4Hz theta waves, a very interesting “water effect” occurred in his brain once he stop listening, as well as increased sensitivity in his senses.

A suggestion might be found in the work of Patterson and Capel (1983) in Surrey, England.

They found that different neurotransmitters were triggered by different frequencies and wave forms. For example, a 10-hertz signal boosts production and turnover rate of serotonin.

"Each brain center generates impulses at a specific frequency, based on the predominant neurotransmitters it secretes," says Dr. Capel. "In other words, the brain's internal communications system--its language, if you like--is based on frequency..." (Ostrander & Schroeder, 1991).

The implications of Capel's & Patterson's work is that one can alter the brain's neuro-chemistry, and thereby it's functioning, with modifications of brain wave frequency.

- 4.11 - Associated with kidneys; strength.
- 4.125 – Associated with DNA restoration; harmonic of the Solfeggio tone **528Hz** (which is the exact frequency used by genetic biochemists to repair broken DNA) and recent Earth resonant frequency shift.
- 4.5 - Associated with Shamanic state of consciousness and Tibetan Buddhist chants.
- 4.6 - Associated with spleen & blood; emotional impulse.
- 4.9 -Associated with introspection; induce relaxation, meditation & deeper sleep.
- 5.0 - Associated with unusual problem solving; less sleep needed; theta sounds replace need for extensive dreaming; relaxed states; pain relief (beta endorphin increases of 10-50% reported); alleged sphincter resonance (??)
- 5.14 - Associated with stomach; emotional acceptance.

- 5.35 - Associated with lungs.
- 5.5 - Moves beyond knowledge to knowing; shows vision of growth needed ("inner voice" guidance; intuition.
- 5.8 – Reduces fear, absent-mindedness, dizziness.
- 6.0 - Associated with long term memory stimulation; reduces unwillingness to work.
- 6.15 - Associated with heart; love,warmth.
- 6.2 – 6.7: Frontal Midline Theta (Fm Theta) is a specific EEG frequency seen in those subjects actively engaged in cognitive activity, such as solving math problems & playing Tetris.
- 6.26 – 6.6: Hemispheric desync, confusion, anxiety, low Reaction Time, depression insomnia.
- 6.30 – Associated with telepathy mental/astral projection; accelerated learning; increased memory retention (??); reduce anger + Irritability (??)
- 6.8 - Associated with telepathy; possible use for muscle spasms.
- 6.88 Associated with collarbones; vitality; overall balance; stability.
- 7.0 – Associated with telepathy mental/astral projection; bending objects; psychic surgery; increased reaction time; mass aggregate frequency (i.e. can de-aggregate matter). Also alleged to resonate/rupture organs at excessive intensity; treatment of sleep disturbances; bone growth.
- 7.69 - Associated with shoulders; strength of the arms; expansion; teaching.
- 7.8 - Associated with ESP activation; Doyere's group (1993), found that short high frequency bursts at 7.7 Hz induced LTP in prefrontal cortex, though only for one day.
- 7.83 - Associated with anti-jetlag, anti-mind control, improved stress tolerance; psychic healing experiments; pituitary stimulation to release growth hormone (helps develop muscle, recover from injuries, rejuvenation effects); reports of accelerated healing/enhanced learning.
- 8.0 – Associated with DNA repair (RAD-6). Also 4Hz is a direct harmonic of 8Hz (covered above).
- 8.22 - Associated with mouth; speech; creativity.

- 8.3 - Pick up visual images of mental objects (??)
- 9.0 - Associated with awareness of body imbalance causes; dirct harmonic of 4.5Hz above.
- 9.19 - Associated with upper lip; emotions; conflict resolution.
- 9.4 - Major frequency used for prostate problems.
- 10 - Enhances release of serotonin; mood elevator; acts as analgesic; supposedly also good for hangovers & jet lag (and nicotine withdrawal, in one case). Decreased pain; arousal, increased alertness and sense of well being; significant improvements in memory, reading and spelling are reported (reportedly in conjunction with 18 HZ; 8Hz beat frequency). Also a direct harmonic of 5Hz covered above.
- 10.2 - Associated with catecholamines.
- 10.3 Associated with nasal passages; breathing; taste.
- 10.5 - Associated with healing of body; stimulating for the immunity.
- 10.6 - Relaxed & alert.
- 10.7 - Associated with ears; hearing; formal concepts.
- 11.0 - Used to achieve "relaxed yet alert" states; see also 5.5Hz entry above, as 11Hz is a direct harmonic.
- 12.3 - Associated with eyes; visualization.
- 13.0 – Alleged sphincter resonance (??).
- 13.8 - Associated with the seventh sense, final decision.
- 14.0 - Alert focusing; vitality; concentration on tasks; audio-visual stimulation alternating between 14 & 22 Hz?
- 15.0 – Allegedly helps alleviate chronic pain.
- 15.4 - Associated with cortex; intelligence.
- 16.0 - Bottom limit of normal hearing; release oxygen & calcium into cells; direct harmonic of both 4HZ & 8Hz (covered above).

- 20.0 – Stimulant; used on sinus disorders/infections, head cold, and headache; direct harmonic of both 5Hz & 10Hz (covered above).
- 20.215 - Allegedly can produce LSD 25-like effects and/or Ψ (??)
- 22.027 – Associated with increased insertion; possibly Ψ (??)
- 25.0 - Bypassing the eyes for image-imprinting (visual cortex); tested clinically with patients who complain of anxiety.
- 31.32 - Pituitary stimulation to release growth hormone (helps develop muscle, recover from injuries, rejuvenation effects).
- 32 - Desensitizer; enhanced vigor & alertness.
- 33 - Associated with Christ consciousness; hypersensitivity; harmonic of 16.5Hz, covered above.
- 38 – Endorphin release.

The Brain's "Sweet Spot" Operating System Frequency

Commonly referred to as the brain's "operating system", **40Hz** activity ranges from **38.8-40.1Hz**; regardless of the electrode location. The average frequency is in the ~39.5-Hz.

In summary, when the body is profoundly relaxed and the mind is in a state of high focus and concentration, 20Hz/40Hz brain activity can be seen in the raw and quantitative EEG of some subjects. Not surprisingly, 40Hz is associated with **being in the Zone**.

It is also associated with information-rich task processing and high-level information processing; dominant in fearful problem-solving situations; synchronization in rapidly processing incoming sensory stimulation, and also a harmonic of 20Hz, 10Hz and 5Hz .

"'For scientists who study the human brain, even its simplest act of perception is an event of astonishing intricacy. 40 Hz brain activity may be a kind of binding mechanism".

- Dr. Rodolfo Llinas, professor of neuroscience (New York University)

Llinas believes that the 40Hz wave serves to connect structures in the cortex (where advanced information processing occurs) and the thalamus, a lower brain region where complex relay and integrative functions are carried out.

In another study quite some time back called *40Hz brain activity, consciousness, and psi* in which gamblers were presented a computer screen with four playing cards shown on them, and then asked to guess which of the four cards would appear on the screen by itself next.

Not surprisingly, their guessing was about as accurate as random chance.

However, *when they had their eyes on the correct card*, the amount of 40Hz activity registering in their brain was significantly higher, and involved in the processing of **Ψ information**:

"EEG data recorded from 22 frequent gamblers while they played a laboratory, video-gambling game. On each hand, four cards were presented sequentially in the center of a video screen using an inter-stimulus interval of 2600 ms and a stimulus duration of 330 ms in counterbalanced wager and non-wager blocks.

Then, subjects guessed which card would later be selected as the target by a computerized, random process. Guessing accuracy was at chance level (,p = .465; one-tailed binomial). EEG analyses focused on 40Hz power in the 150-500 msec latency window following delivery of the target and non-target cards.

ANOVAs indicated greater 40Hz power for targets than for non-targets over left-frontal scalp in both wager and non-wager conditions, F (1/21) = 7.87, p = .005, and over the right-posterior scalp in the wager condition, F (1/21) = 5.81, p = .013, both one-tailed.

These findings indicate that:

1) 40Hz activity is involved in the processing of psi [ψ] information.

2) The attentional mechanisms of the brain utilized in focused arousal are also utilized in processing psi [ψ] information in the wager condition, although the left-frontal effect, found in both wager and non-wager conditions, is unlike the posterior-parietal locus of 40Hz effects observed by Sheer.

Neither the left frontal nor the right parietal loci of effects overlay primary sensory areas of the cerebral cortex, suggesting that the 40Hz effects we observed were not due to sensory processes.

The present EEG findings are consistent with the notion of unconscious or preconscious psi. Guessing accuracy was at chance level and yet 40Hz activity following target and non-target stimuli differed significantly.

Thus, although conscious behavior was not influenced by psi information, the differential brain responses indicated recognition at some level that the stimuli belonged to different categories. These findings in the frequency domain support and extend our previous findings with event-related brain potentials."

If the scientific aspect of the 40Hz “brain OS” fascinates you, this page cites a wealth of interesting studies and information.

40 Hz Gamma Pure Tone Binaural Beat: https://www.youtube.com/watch?v=ZGHbKWGgH_E

Pt. 3: Cossum's Awesome Enhanced Brain Machine

"I can see through mountains, watch me disappear, I can even touch the sky.
Swallowing colors of the sound I hear, am I just a crazy guy? You bet!"

- Ozzy Osbourne/"Flying High Again"

Image via Makezine.com

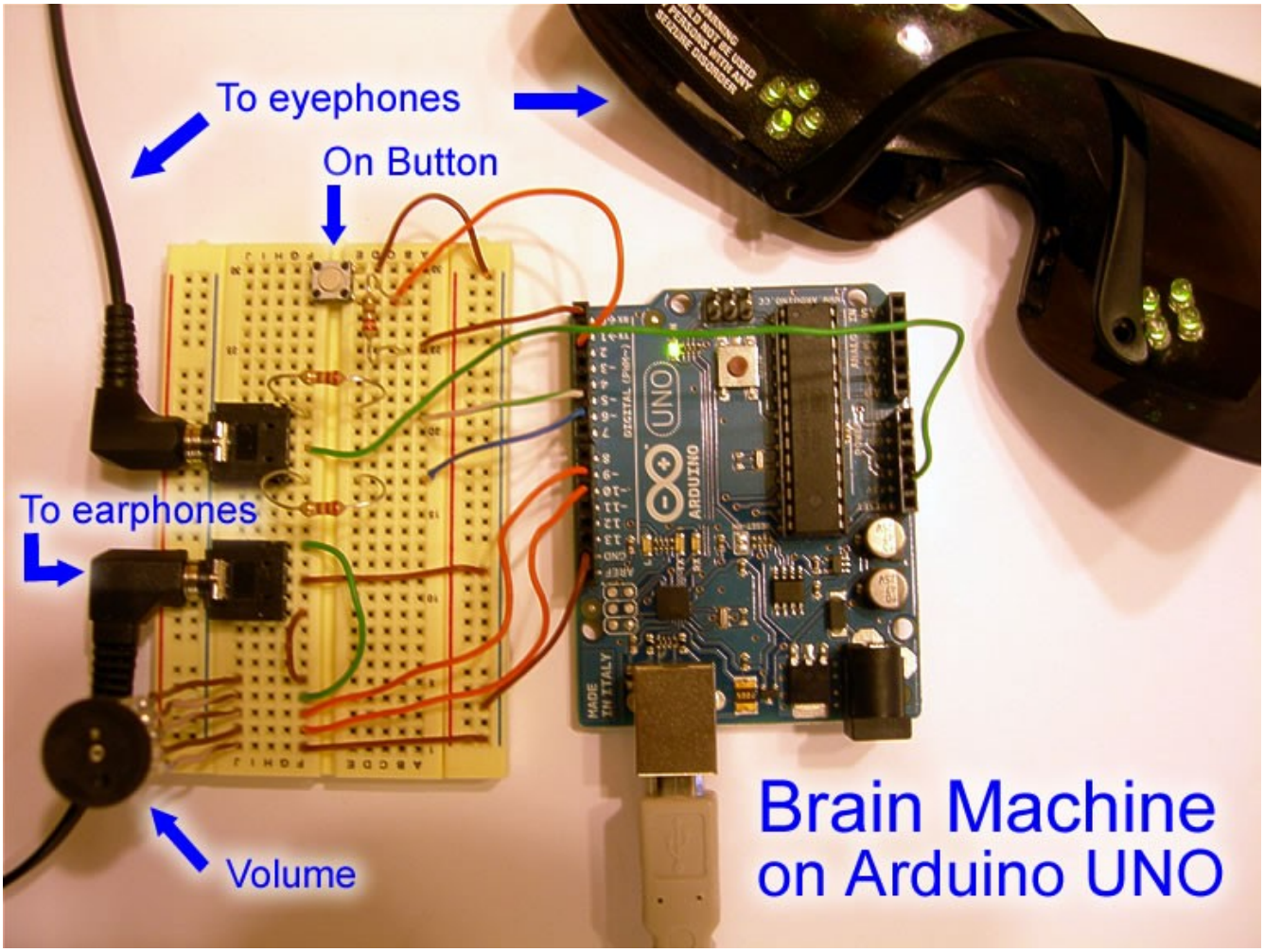
Attribution: The following adaptation by Mark Maffei is based on Bobby Cossum's documentation, of which both his documentation and code are licensed GPL; hence this adaptation is via CC BY SA, *provided this entire attribution remains fully intact*. However, you can slightly reword it, to fit the flow of your remix.

Brief SLM Introduction & Historical Tidbits

Historically, microcontroller-based sound/light machines (SLM's) really took off in 2007 via Mitch Altman and his famous hack of Lady Ada's MiniPOV kit, which he simply called the "Brain Machine". Here's a short but interesting interview of Altman: https://www.youtube.com/watch?v=vysfoePdiik

Based on the ATtiny2313P, It was an instant hit after being featured in **Make: 10**; with thousands of Makers from around the world experiencing the "forbidden pleasures" of neurohacking for the very first time.

As to be expected, several custom mods based on his initial design sprouted up like magic mushrooms over the years. One such derivative of *historical significance* was a full porting over to UNO by Chris, back in early 2011:



The Code:

Here's Chris' revised code and documentation for his UNO-based version, with several subtle, yet interesting improvements on Altman's original ATtiny Brain Machine design:

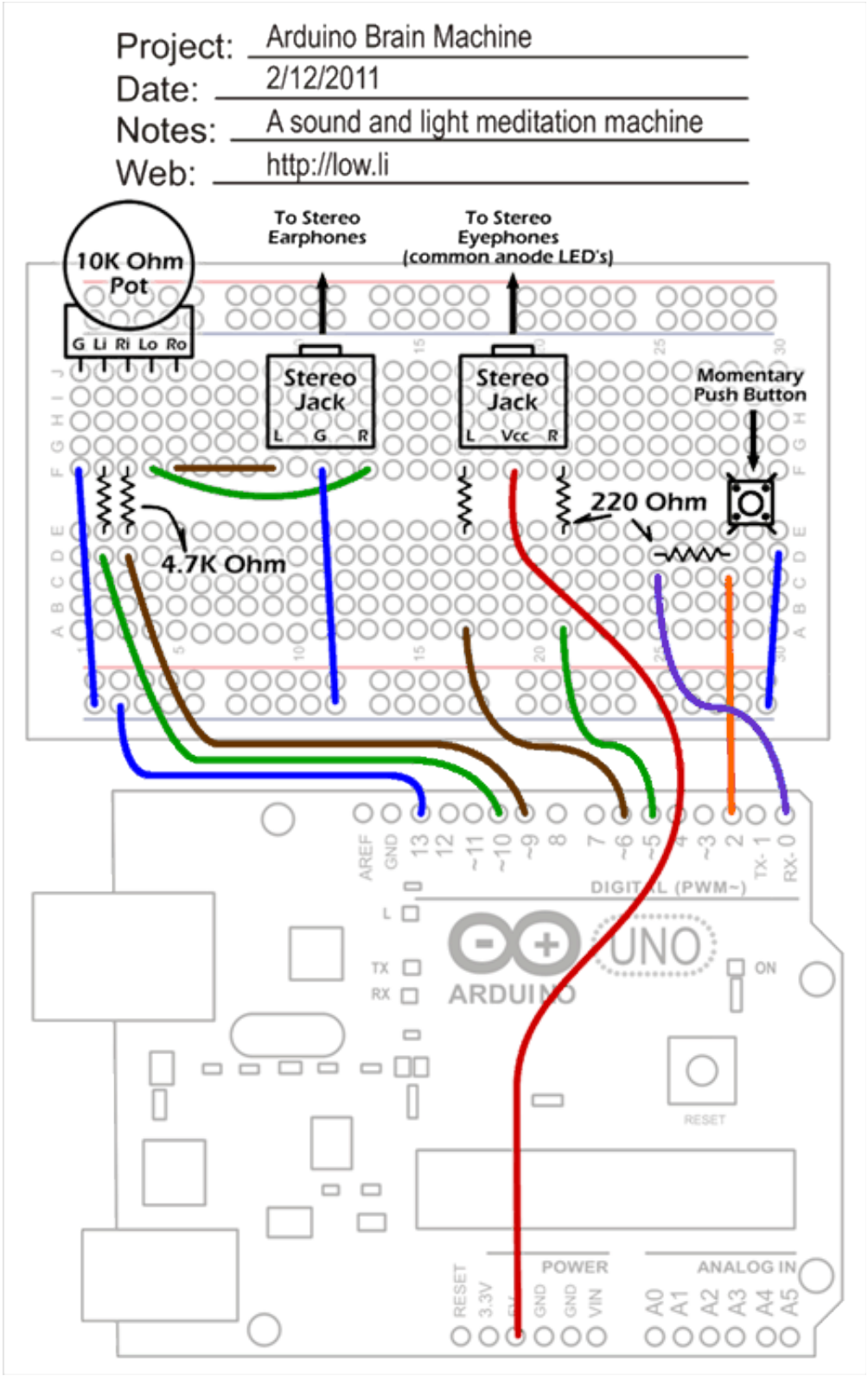
https://github.com/LaughterOnWater/Arduino-Brain-Machine

And while both Altman's and Lady Ada's projects (in their original form) have long since been depreciated... Altman's original Brain Machine nonetheless lives on as the "Godfather" of all modern Arduino-based SLM's.

However, of *equally great historical relevance is the amazingly awesome upgrades* **Bobby Cossum** unveiled at the 2010 Maker Faire; directly based on Mitch Altman's original Brain Machine design.

But first, for archival reference here's...

Chris' UNO-Based Brain Machine Schematic:



Cossum's Awesome Enhanced Brain Machine Upgrades

Three of Cossum's upgrades that are especially noteworthy:

1. It was the first 100% open source **RGB-based SLM**;

2. It incorporates a brilliant method of turning the annoying-sounding previous square waves generated into *far more pleasant sounding* **pseudo-sine waves**;

3. Plus, his enhancements are compatible with both the original ATtiny-based version **and Arduino**.

Unfortunately, the code and documentation is spread out across multiple pages…

That is, until now! I have gone to *great lengths* to faithfully reconstruct Cossum's most excellent contribution to the Brainhacker Community below, in an easy-to-follow manner.

[**Editor Note:** Cossum's original documentation/notes will be in this font; editing as necessary for clarity/flow. Also, the **ATtiny-based** enhancements/mods directly correspond to Altman's original schematic below]

Cossum's SLM Brain Machine Documentation Remixed

An RGB Brain Machine! A screen saver for your head! Mitch Altman's Brain Machine from Make: 10 redesigned with sine wave-ish audio and RGB LEDs. This project can be built on a MiniPOV3 or on an Arduino. To build the code for a MiniPOV3, run the makefile in the archive. To build for an Arduino, unzip the archive into an Arduino project directory and rename the file slm.c to slm.pde.

[Download The Code]

Remixed From The Associated Wiki Page:

Introduction

This page describes an upgrade to the Brain Machine project featured in Make: 10. By applying the hardware and firmware modifications described, you will end up with a device with superior audio and firmware controllable LED brightness.

The updated firmware provides more pleasing sine wave-like audio using a technique known as direct digital synthesis. With the supplied firmware, beat frequencies from 14Hz down to 1Hz are available in 1Hz increments; with a base frequency of 200Hz. However, the code is easily modifiable; *no longer requiring hairy timer math* as with the original firmware.

Details

Tools Required:

- Soldering iron
- Desoldering braid
- Solder

Hardware Modification:

The hardware modification involves the removal of R4, R6, LED5 and the capacitor nearer th DB-9 connector. R6 and the capacitor will be re-used so try to remove them without damaging them. R4 and R6 are both labeled on the circuit board. LED5 is next to the capacitor that is not being removed.

Further:

Further and more interesting modifications to your Brain Machine are possible. The addition of RGB LEDs adds a whole new dimension to the experience. Unfortunately blue and green LEDs require a voltage higher than the 3 volts available from two AA alkaline batteries or the 2.4 volts from a pair of nickel metal hydride cells.

Remixed From The “README” File Contained Within slm.zip:

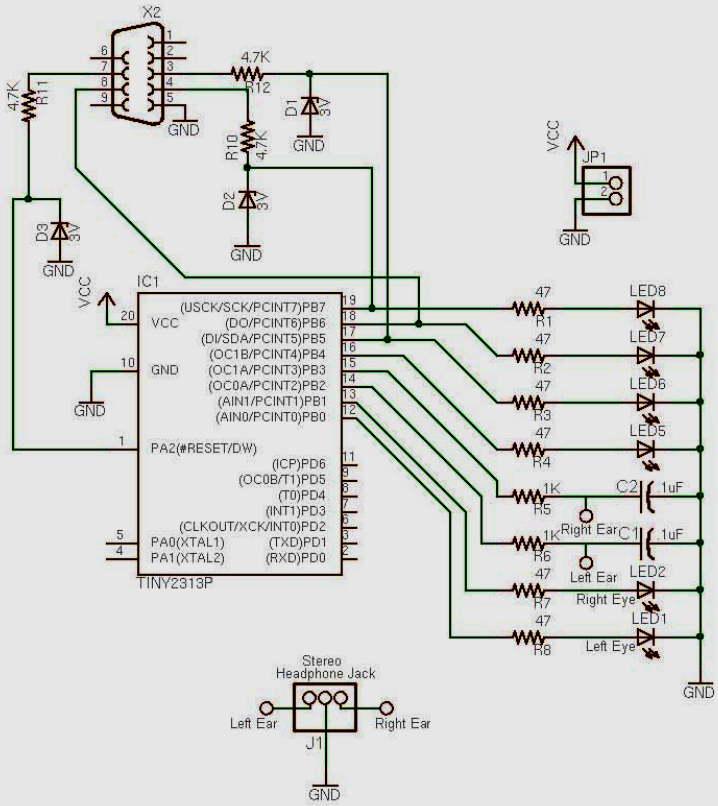
This file describes the process for rebuilding the Brain Machine firmware with your own brainwave sequences. All of the data for the brainwave sequence is contained in the file table.h and table.h is the only file you should attempt to edit unless you feel yourself to be a competent C programmer and are willing to risk breaking the code.

If you do break the code, you can download a clean copy from:
TheEntropyWorks.googlecode.com/files/slm.zip

Note that this firmware will not run on the hardware described by Mitch Altman in MAKE #10. There are small but significant differences in how the circuit is built. The connections required are described in comments in the file slm.c.

First, let's discuss the construction of the brainwave table. It is a sequence of numerical pairs specifying an integer beat frequency from 1 to 15 beats per second and a duration in seconds. The end of the table is an entry with a beat frequency of 0.

Mitch Altman's Original "Brain Machine"



R5 / C2 and R6 / C1 are each Low Pass Filters
for converting the square-wave output of the
microcontroller to sine-waves (more or less).

Here's the math:
We want the cut-off frequency (Fc) of the Low Pass Filter < half 400Hz tone we'll be using:
Fc = 1 / (2 * Pi * R * C) = 1 / (6.28 * 1000 * .000001) = 160Hz
This works because a 400Hz square-wave can be modeled as a 400Hz sine-wave
with lots of multiples of this frequency added to it.
The LPF greatly diminishes the volume of all frequencies greater than 400Hz,
leaving the 400Hz sine-wave mostly untouched.

Distributed under Creative Commons 2.5 - Attrib. & Share Alike	
TITLE: SLM for MAKE (modified from minipov3)	
Document Number: modified from original dwg at: http://www.ladyada.net/make/minipov3	REV:
Date: 1/31/2007	Sheet: 1/1

To begin the brainwave table type the following:

BW_TABLE_BEGIN
This should be followed by a number of lines that look like this:

BW_TABLE_ENTRY (14, 60)

This specifies a 14 beat per second beta frequency for 60 seconds.

At the end of the sequence include the following:

BW_TABLE_END

The brainwave table is really all you need to specify, but there are a number of other things that you can control to customize your experience.

You can specify a base frequency for the audio part of the Brain Machine. by default this is 200Hz. I have not played around with this very much, but to change it from the default enter something like the following, which will set the base frequency to 400Hz:

BASE_FREQUENCY (400)

It is also possible to customize the LEDs. This can provide some interesting effects. By default, the red LEDs flash while the blue and green LEDs fade in and out creating a moving field of color as a background.

There are a few constants already defined so that you can specify LEDs. These are:

RED: Specifies both left and right red LEDs
LEFTRED: Left red LED only
RIGHTRED: Right LED led only
GREEN: Specifies both left and right green LEDs
LEFTGREEN: Left green LED only
RIGHTGREEN: Right green LED only
BLUE: specifies Both left and right blue LEDs
LEFTBLUE: Left blue LED only
RIGHTBLUE: Right blue LED only

Then there are a number of constants that you can define using the constants above to control the behavior of the LEDs. These are:

FLASH: Chose which LEDs will flash
FADE1: These LEDs will fade in while the FADE2 LEDs fade out
FADE2: these LEDs will fade in while the FADE1 LEDs fade out

To set these constants you must use the C compiler "define" directive. By flashing the red LEDs and leaving the blue and green LEDs off, you can emulate the Brain Machine in

Make: 10. The syntax for this is:

#define FLASH RED
#define FADE1 0
#define FADE2 0

My favorite configuration is to flash the blue LEDs and have the red and green LEDs fade back and forth between the right and left eyes. This gives a less harsh flash with a warm moving background. To get this effect use the following:

#define FLASH BLUE
#define FADE1 LEFTRED | RIGHTGREEN
#define FADE2 LEFTGREEN | RIGHTRED

The vertical bar in the statements above is a logical or operand. It is used to add multiple LEDs to a channel. Adding the same LED to FADE1 and FADE2 makes no sense and will generate an error.

Adding the same LED to either FADE1 or FADE2 and to FLASH will also generate an error, but can be enabled by including the following line:

#define UNSTRICT

Here is an effect that I haven't yet tried but mean to. To have flashing LEDs that fade from red to magenta to blue with the colors cross-fading, so that when one eye LED is red, the other LED is blue, use the following:

#define UNSTRICT
#define FLASH RED | BLUE
#define FADE1 LEFTRED | RIGHTBLUE
#define FADE2 RIGHTRED | LEFTBLUE

The last element you can control is LED intensity. There are three parameters you can define:

FADEMAX: The maximum intensity for LED fades.
FADEMIN: The minimum intensity for LED fades.
FLASHINT: The intensity of the flashing LEDs.

Legitimate values for these are in the range of 0 to 127. It is possible to use values outside of this range but the LED behavior will be unpredictable. It is also possible to specify a fade minimum greater than the fade maximum; and although the behavior in this case will be predictable, I don't care to try to describe it.

So what are reasonable values? I like to set the flash intensity to the maximum (127), so that it will be visible; the fade maximum to 63 so that it doesn't wash out the flashing, and the fade minimum to 0.

Note that in the UNSTRICT example above, the lesser of flash intensity and fade maximum will be the value that is used. I would probably use 127.

Here is an example of a complete table.h file:

```
#define FLASHINT 127
#define FADEMAX 63
#define FADEMIN 0
#define FLASH RED
#define FADE1 LEFTBLUE | RIGHTGREEN
#define FADE2 LEFTGREEN | RIGHTBLUE

BASE_FREQUENCY (200)

BW_TABLE_BEGIN
  BW_TABLE_ENTRY (14, 60)
  BW_TABLE_ENTRY (11, 10)
  BW_TABLE_ENTRY (14, 20)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY (14, 15)
  BW_TABLE_ENTRY (11, 20)
  BW_TABLE_ENTRY (14, 10)
  BW_TABLE_ENTRY (11, 30)
  BW_TABLE_ENTRY (14, 5)
  BW_TABLE_ENTRY (11, 60)
  BW_TABLE_ENTRY ( 7, 10)
  BW_TABLE_ENTRY (11, 30)
  BW_TABLE_ENTRY ( 7, 20)
  BW_TABLE_ENTRY (11, 30)
  BW_TABLE_ENTRY ( 7, 30)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY ( 7, 60)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY (14, 1)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY ( 7, 60)
  BW_TABLE_ENTRY ( 2, 1)
  BW_TABLE_ENTRY ( 7, 10)
  BW_TABLE_ENTRY ( 2, 1)
  BW_TABLE_ENTRY ( 7, 10)
  BW_TABLE_ENTRY ( 2, 1)
  BW_TABLE_ENTRY ( 7, 30)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY (14, 1)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY ( 7, 30)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY (14, 1)
  BW_TABLE_ENTRY (11, 20)
  BW_TABLE_ENTRY (14, 5)
```

```
  BW_TABLE_ENTRY (11, 20)
  BW_TABLE_ENTRY (14, 15)
  BW_TABLE_ENTRY (11, 15)
  BW_TABLE_ENTRY (14, 20)
  BW_TABLE_ENTRY (11, 10)
  BW_TABLE_ENTRY (14, 25)
  BW_TABLE_ENTRY (11, 5)
  BW_TABLE_ENTRY (14, 60)
BW_TABLE_END
```

Code Comments (From Within table.h) Remixed

[**Editor Note:** The following code comments from Cossum are masterfully remixed for brevity and ease of reading, so as to *stay in context* with the above documentation]

This implementation attempts to synthesize a sine wave-like audio output more pleasing to the ear and uses RGB LEDs. The firmware is intended to blink one color while fading the others in and out, creating a moving field of color behind.

This effect was originally devised to test my code for PWM control of the LEDs, but I found it pleasing and left it in. Other effects such as blinking different colors at different beat frequencies, or having each color beat simultaneously at different frequencies should be trivial modifications to the code.

This software will run on an Arduino/ATmegaX68 (8MHz or 16MHz) with the following connections:

PB1 - base frequency audio channel
PB2 - offset frequency audio channel
PD2,PD5 - red LEDs
PD3,PD6 - green LEDs
PD4,PD7 - blue LEDs

For the Arduino, that's:

D9 - base frequency audio channel
D10 - offset frequency audio channel
D2,D5 - red LEDs
D3,D6 - green LEDs
D4,D7 - blue LEDs

It will also run on an ATtiny2313, specifically a modified Adafruit Industries MiniPOV3 kit clocked at either 8MHz or 16MHz. The connections to the ATtiny2313 should be:

PB3 - base frequency audio channel
PB4 - offset frequency audio channel
PB0,PB5 - red LEDs

PB1,PB6 - green LEDs
PB2,PB7 - blue LEDs

Note that these connections differ from Mitch's version most significantly in that one of the audio channels is moved from PB2 to PB4 so that it can be connected to OC1B output of timer 1.

The LED outputs can be moved around freely by modifying symbols defined in the file slm.h. The green and blue outputs need not really be connected. By defining the symbols RRED and LRED in the file slm.h as PB0 and PB1, one can convert an existing SLM with the only hardware modification being the movement of one audio channel.

How's It Work?

All of the outputs are generated in the timer 1 overflow interrupt service routine. The only things that happen in the main program are hardware initialization, setting up the brainwave sequence, then looping waiting for the sequence to end (while periodically changing the brightness of the green and blue LEDs), and finally stopping timer 1, turning off all the outputs and putting the MCU to sleep.

Timer 1 overflows every 512 ticks of the MCU's clock. At 8MHz this results in an interrupt frequency of 15.625KHz, at 16MHz it's 31.250KHz.

The advantage of running at 16MHz is that the audio PWM frequency will be well above the range of human hearing. At 8MHz one must rely on a low pass filter, cheap headphones and high frequency hearing loss.

At each timer interrupt, a pulse is output on OC1A with a width proportional to the desired amplitude of the base frequency sine wave. The offset frequency is output as a pulse on OC1B.

For each channel, the integer part of a 24 bit fixed point (8 bit integer, 16 bit fraction), phase accumulator is used as an index into a 256 byte table of 8 bit amplitudes. For each entry [i] in the amplitude table, sinetab[i] = 127 + sin(i * pi / 128) * 127.

For each frequency to be output, a phase step is calculated at compile time. This step is the change in phase for the frequency at each interrupt and is calculated by multiplying the frequency by the size of the sine wave table and dividing by the interrupt frequency. Step sizes for the base frequency and offset frequencies from 1 to 15 Hz above the base frequency are stored in a lookup table.

The flash rate of the LEDs is controlled by comparing the phases of the offset and base waveforms. The sign of the result of offset phase - base phase determines the state of the LEDs. Color fading is done with seven bit PWM. So the interrupt frequency of 15.625KHz is divided by 128 for a PWM frequency of ~120Hz. At every interrupt a counter is compared to 128.

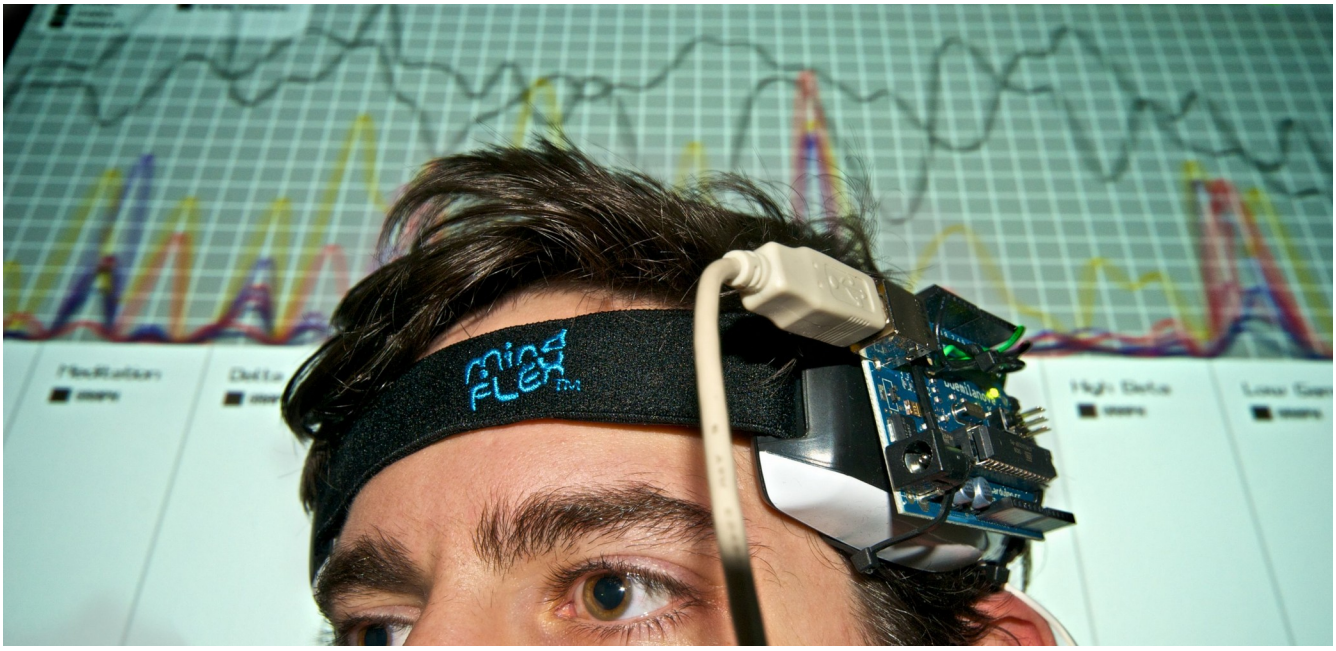
When it reaches 128, it is reset to zero, the current values of all colors are copied to locations to be used for comparisons during the current PWM period, and all the LEDs are turned on. At every interrupt, the stored color values are compared to the counter and when they are equal the corresponding LEDs are turned off. The counter is then incremented.

Pt. 4: The Art Of EEG Toy Hacking Gone Wild!

How To Hack Toy EEGs

Attribution: "*How To Hack Toy EEGs*" by direct permission from Eric Mika to specifically be included in Ultimate Arduino Brainhackar's Handbook via CC BY SA.

Eric's Home Page: EricMika.com



Arturo Vidich, Sofy Yuditskaya, and I needed a way to read brains for our Mental Block project last fall. After looking at the options, we decided that hacking a toy EEG would be the cheapest / fastest way to get the data we wanted. Here's how we did it.

The Options

A non-exhaustive list of the consumer-level options for building a brain-computer interface:

Open EEG offers a wealth of hardware schematics, notes, and free software for building your own EEG system. It's a great project, but the trouble is that the hardware costs add up quickly, and there isn't a plug-and-play implementation comparable to the EEG toys.

The Neurosky MindSet is a reasonable deal as well — it's wireless, supported, and plays nicely with the company's free developer tools.

For our purposes, though, it was still a bit spendy. Since NeuroSky supplies the EEG chip and hardware for the Force Trainer and Mind Flex toys, these options represent a cheaper (if less convenient) way to get the same data.

The silicon may be the same between the three, but our tests show that each runs slightly different firmware which accounts for some variations in data output. The Force Trainer, for example, doesn't output EEG power band values — the Mind Flex does. The MindSet, unlike the toys, also gives you access to raw wave data.

However, since we'd probably end up running an FFT on the wave anyway (and that's essentially what the EEG power bands represent), we didn't particularly miss this data in our work with the Mind Flex.

Comparison At A Glance:

	 Open EEG	 Force Trainer	 Mind Flex	 MindSet
Description	Plans and software for building an EEG from scratch	Levitating ball game from Uncle Milton	Levitating ball game from Mattel	Official headset from NeuroSky
Attention / Meditation Values	No	Yes	Yes	Yes
EEG Power Band Values	Yes (roll your own FFT)	No	Yes	Yes
Raw wave values	Yes	No	No	Yes
Cost	$200+	$75 (street)	$80 (street)	$200

Given all of this, I think the Mind Flex represents a sweet spot on the price / performance curve.

It gives you almost all of the data the Mind Set for less than half the cost. The hack and accompanying software presented below works fine for the Force Trainer as well, but you'll

end up with less data since the EEG power values are disabled in the Force Trainer's firmware from the factory.

Of course, the Mind Flex is supposed to be a black-box toy, not an officially supported development platform — so in order to access the actual sensor data for use in other contexts, we'll need to make some hardware modifications and write some software to help things along. Here's how.

But first, the inevitable caveat: *Use extreme caution when working with any kind of voltage around your brain, particularly when wall power is involved. The risks are small, but to be on the safe side* **you should only plug the Arduino + Mind Flex combo into a laptop running on batteries alone**.

(My thanks to Viadd for pointing out this risk in the comments.) Also, performing the modifications outlined below means that you'll void your warranty. If you make a mistake you could damage the unit beyond repair. The modifications aren't easily reversible, and they may interfere with the toy's original ball-levitating functionality.

However, I've confirmed that when the hack is executed properly, the toy will continue to function — and perhaps more interestingly, you can skim data from the NeuroSky chip without interfering with gameplay. In this way, we've confirmed that the status lights and ball-levitating fan in the Mind Flex are simply mapped to the "Attention" value coming out of the NeuroSky chip.

The Hardware

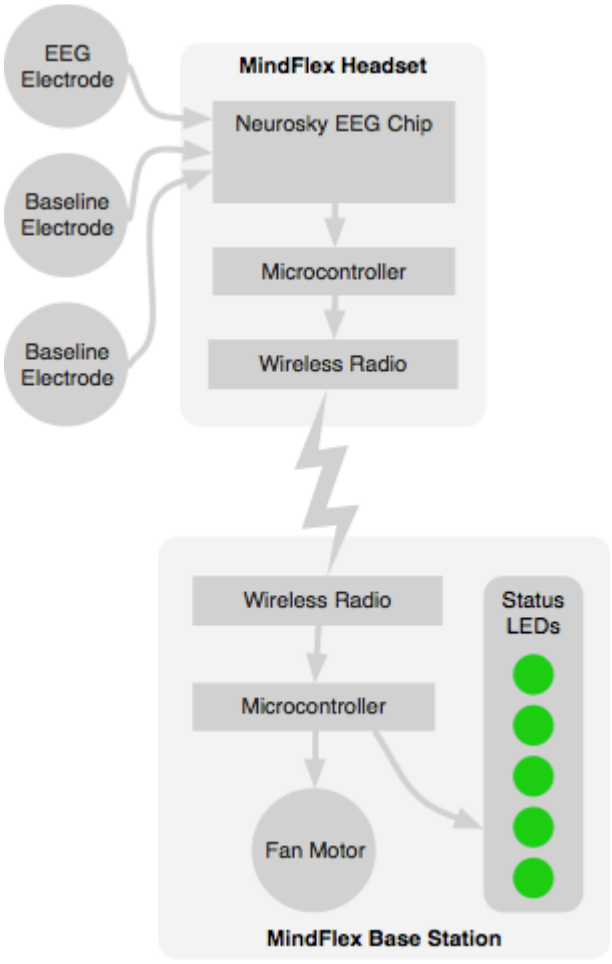
Here's the basic layout of the Mind Flex hardware. Most of the action is in the headband, which holds the EEG hardware.

A micro controller in the headband parses data from the EEG chip and sends updates wirelessly to a base station, where a fan levitates the ball and several LEDs illuminate to represent your current attention level.

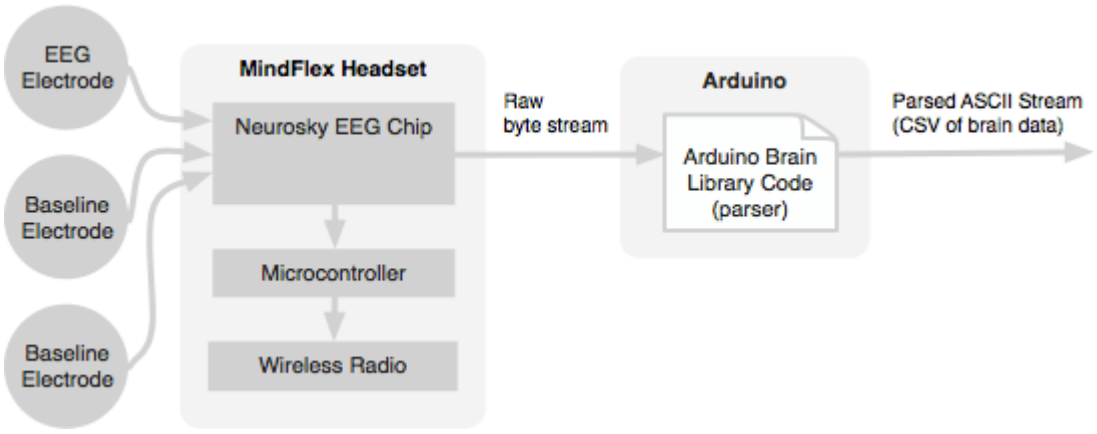
This schematic below immediately suggests several approaches to data extraction. The most common strategy we've seen is to use the LEDs on the base station to get a rough sense of the current attention level.

This is nice and simple, but five levels of attention just doesn't provide the granularity we were looking for.

A Quick Aside: *Unlike the Mind Flex, the Force Trainer has some header pins (probably for programming / testing / debugging) which seem like an ideal place to grab some data. Others have reported success with this approach. We could never get it to work.*



We decided to take a higher-level approach by grabbing serial data directly from the NeuroSky EEG chip and cutting the rest of the game hardware out of the loop, leaving a schematic that looks more like this:



The Hack

Parts List:

- 1 x Mind Flex
- 3 x AAA batteries for the headset
- 1 x Arduino (any variety), with USB cable
- 2 x 12” lengths of solid core hookup wire (around #22 or #24 gauge is best)
- A PC or Mac to monitor the serial data

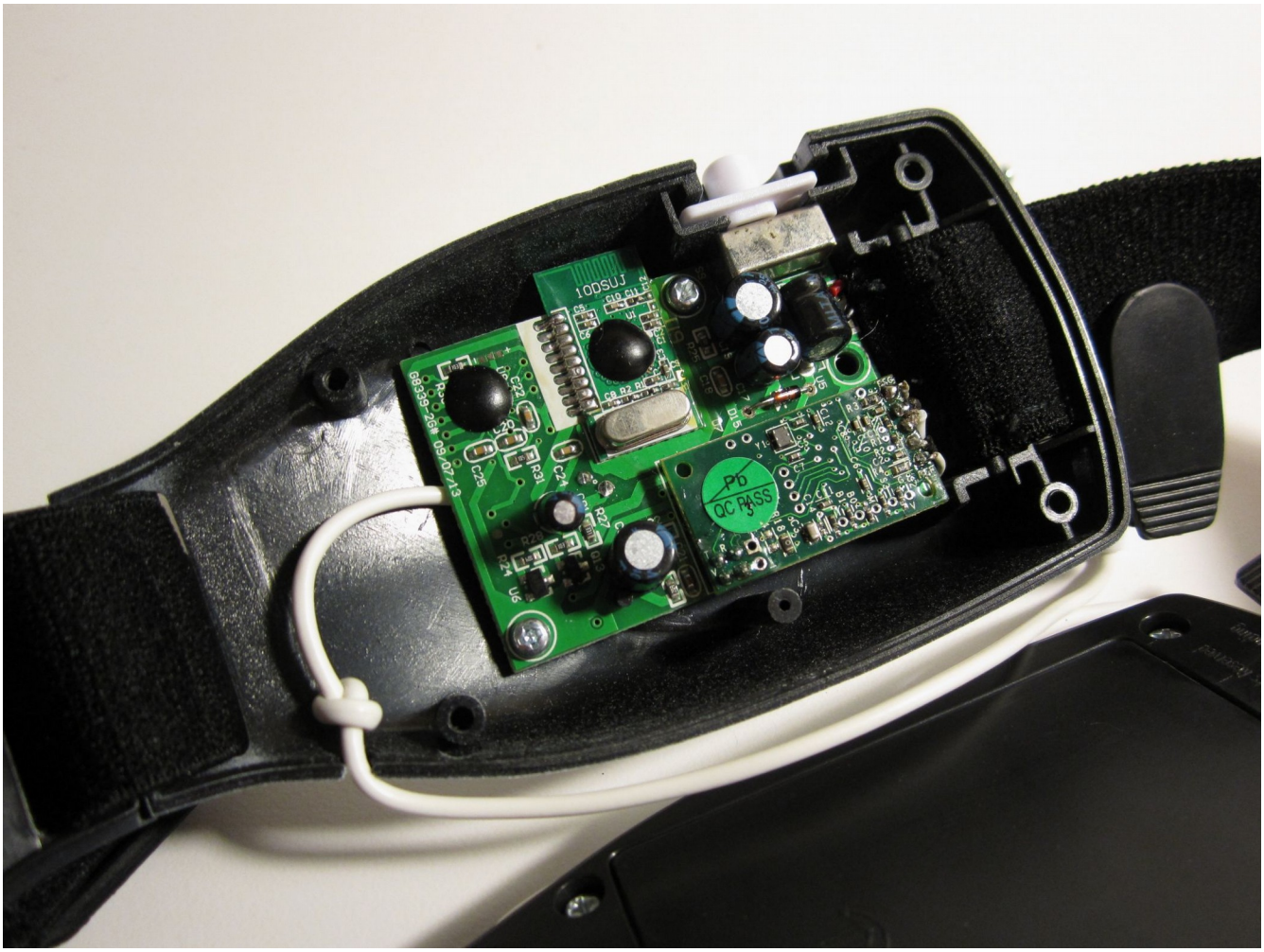
Software List:

- Arduino Brain Library (download here)
- Optional: Processing Brain Visualizer (download here, it will help to have Processing as well)
- Optional (required for the visualizer): controlP5 Processing GUI Library (download here)

The video below walks through the whole process. Detailed instructions and additional commentary follow after the video: https://vimeo.com/10184668

Step-By-Step

1. Disassembly.



Grab a screwdriver and crack open the left pod of the Mind Flex headset. (The right pod holds the batteries.)

2. The T Pin.

The NeuroSky Board is the small daughterboard towards the bottom of the headset.

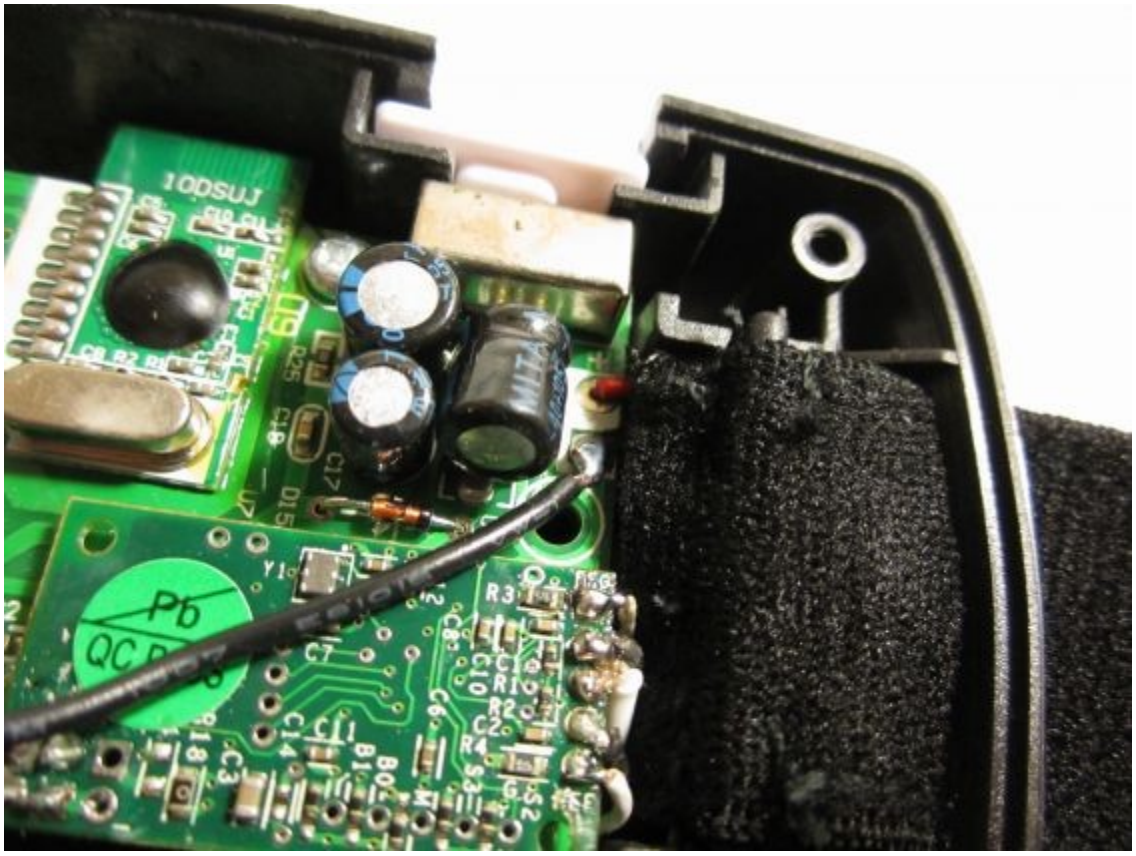
If you look closely, you should see conveniently labeled **T** and **R** pins — these are the pins the EEG board uses to communicate serially to the microcontroller on the main board, and they're the pins we'll use to eavesdrop on the brain data.

Solder a length of wire (carefully) to the "T" pin. Thin wire is fine, we used #24 gauge. Be careful not to short the neighboring pins.

3. Common ground.

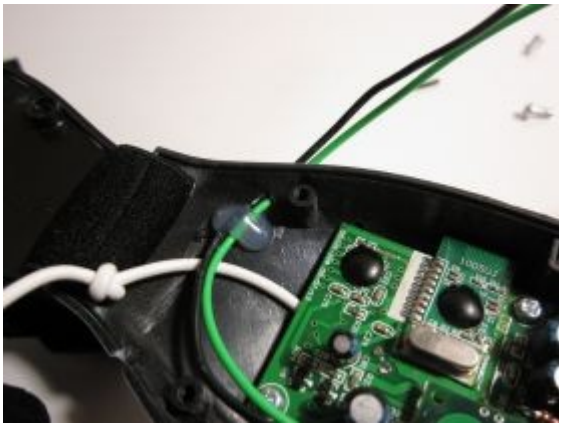
Your Arduino will want to share ground with the Mind Flex circuit. Solder another length of wire to ground — any grounding point will do, but using the large solder pad where the battery's ground connection arrives at the board makes the job easier.

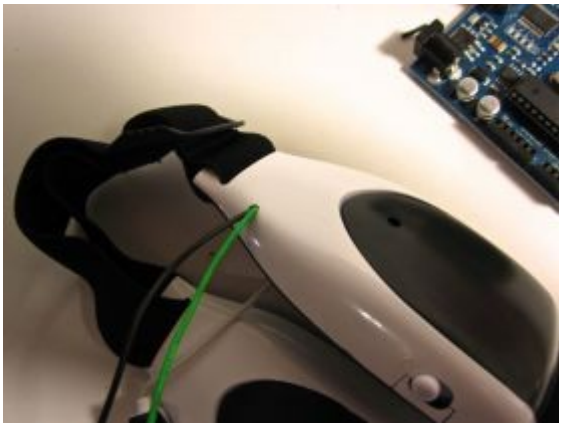
A note on power: *We've found the Mind Flex to be inordinately sensitive to power… our initial hope was to power the NeuroSky board from the Arduino's 3.3v supply, but this proved unreliable. For now we're sticking with the factory configuration and powering the Arduino and Mind Flex independently.*



4. Strain relief and wire routing.

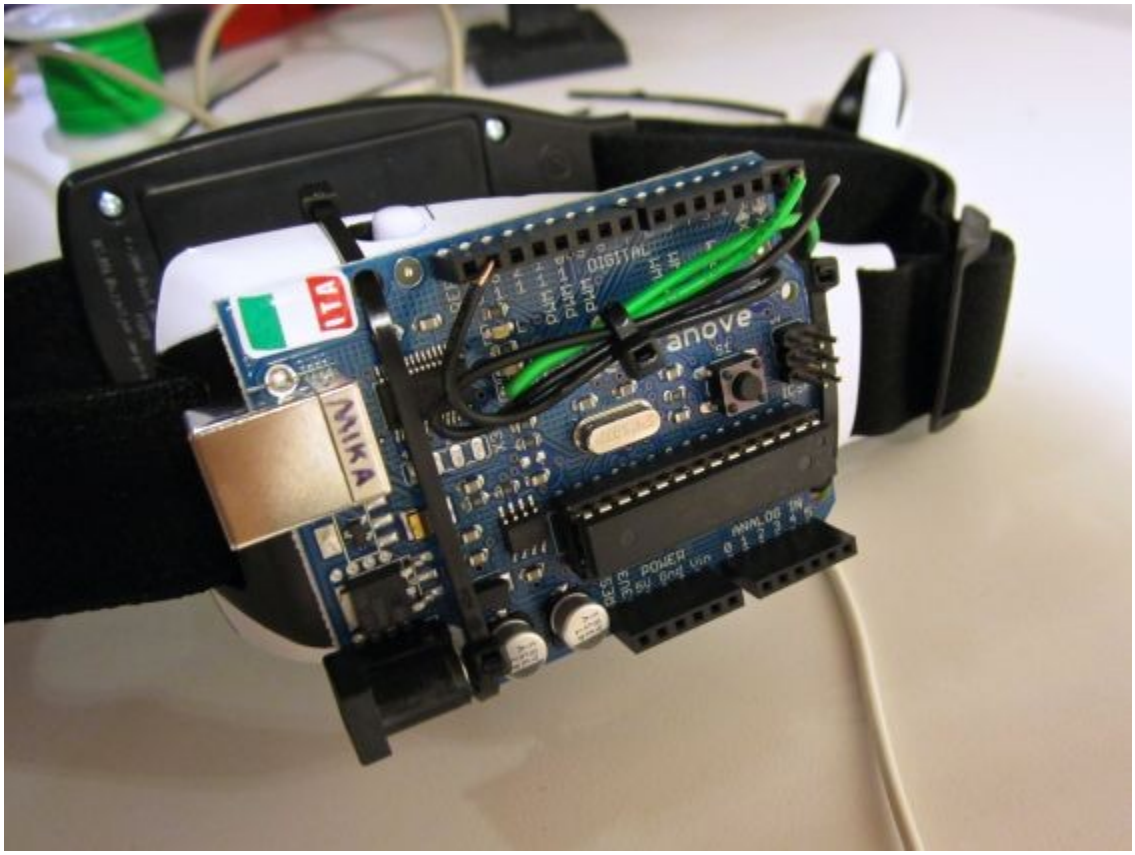
We used a dab of hot glue to act as strain relief for the new wires, and drilled a hole in the case for the two wires to poke through after the case was closed. This step is optional.





5. Hook up the Arduino.

The wire from the Mind Flex's "T" pin goes into the Arduino's **RX pin**. The ground goes… to ground. You may wish to secure the Arduino to the side of the Mind Flex as a matter of convenience. (We used zip ties.)



That's the extent of the hardware hack. Now on to the software.

The data from the NeuroSky is not in a particularly friendly format. It's a stream of raw bytes that will need to be parsed before they'll make any sense. Fate is on our side: the

packets coming from the Mind Flex match the structure from NeuroSky's official Mindset documentation. (See the *mindset_communications_protocol.pdf* document in the Mindset developer kit if you're interested.)

You don't need to worry about this, since I've written an Arduino library that makes the parsing process as painless as possible. Essentially, the library takes the raw byte data from the NeuroSky chip, and turns it into a nice ASCII string of comma-separated values.

6. Load up the Arduino.

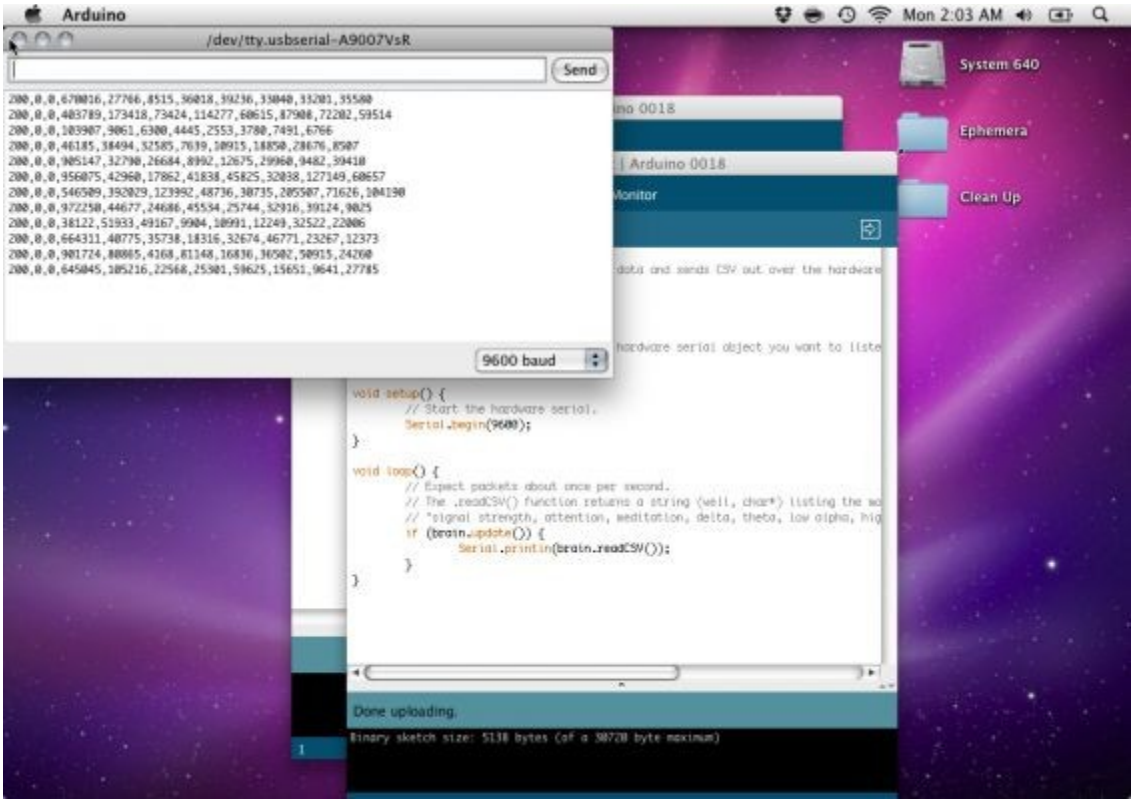
Download and install the Arduino Brain Library — it's available here. Open the *BrainSerialOut* example and upload it to your board. (You may need to disconnect the RX pin during the upload.)

The example code looks like this:

```
#include <Brain.h>

// Set up the brain parser, pass it the hardware serial object you want to
listen on.
Brain brain(Serial);

void setup() {
        // Start the hardware serial.
        Serial.begin(9600);
}

void loop() {
        // Expect packets about once per second.
        // The .readCSV() function returns a string (well, char*) listing the
most recent brain data, in the following format:
        // "signal strength, attention, meditation, delta, theta, low alpha,
high alpha, low beta, high beta, low gamma, high gamma"
        if (brain.update()) {
                Serial.println(brain.readCSV());
        }
}
```

7. Test.

Turn on the Mind Flex, make sure the Arduino is plugged into your computer, and then open up the Serial Monitor. If all went well, you should see the following:



Here's how the CSV breaks down: "signal strength, attention, meditation, delta, theta, low alpha, high alpha, low beta, high beta, low gamma, high gamma"

(More on what these values are supposed to mean later in the article. Also, note that if you are hacking a Force Trainer instead of a Mind Flex, you will only see the first three values — signal strength, attention, and meditation.)

If you put the unit on your head, you should see the "signal strength" value drop to 0 (confusingly, this means the connection is good), and the rest of the numbers start to fluctuate.

8. Visualize.

As exciting as the serial monitor is, you might think, "Surely there's a more intuitive way to visualize this data!" You're in luck: I've written a quick, open-source visualizer in Processing which graphs your brain activity over time (download). It's designed to work with the *BrainSerialOut* Arduino code you've already loaded.

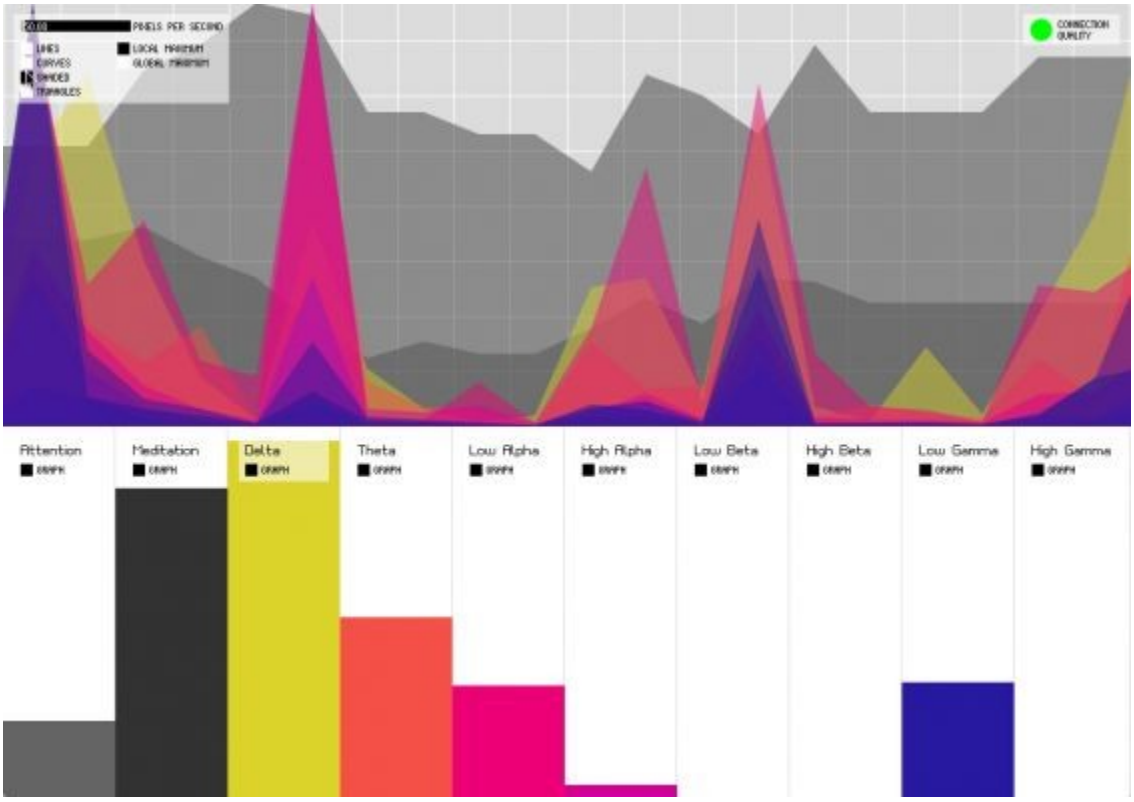
Download the code, and then open up the *brain_grapher.pde* file in Processing.

With the Mind Flex plugged in via USB and powered on, go ahead and run the Processing sketch. (Just make sure the Arduino IDE's serial monitor is closed, otherwise Processing won't be able to read from the Mind Flex.)

You may need to change the index of the serial list array in the brain_grapher.pde file, in case your Arduino is not the first serial object on your machine:

```
serial = new Serial(this, Serial.list()[0], 9600);
```

You should end up with a screen like this:



About The Data

So what, exactly, do the numbers coming in from the NeuroSky chip mean?

The Mind Flex (but not the Force Trainer) provide eight values representing the amount of electrical activity at different frequencies. This data is heavily filtered / amplified, so where a conventional medical-grade EEG would give you absolute voltage values for each band, NeuroSky instead gives you relative measurements which aren't easily mapped to real-world units.

A run down of the frequencies involved follows, along with a grossly oversimplified summary of the associated mental states.

- Delta (1-3Hz): sleep
- Theta (4-7Hz): relaxed, meditative

- Low Alpha (8-9Hz): eyes closed, relaxed
- High Alpha (10-12Hz)
- Low Beta (13-17Hz): alert, focused
- High Beta (18-30Hz)
- Low Gamma (31-40Hz): multi-sensory processing
- High Gamma (41-50Hz)

In addition to these power-band values, the NeuroSky chip provides a pair of proprietary, black-box data values dubbed “attention” and “mediation”.

These are intended to provide an easily-grokked reduction of the brainwave data, and it’s what the Force Trainer and Mind Flex actually use to control the game state. We’re a bit skeptical of these values, since NeuroSky won’t disclose how they work, but a white paper they’ve released suggests that the values are at least statistically distinguishable from nonsense.

Here’s the company line on each value:

- **Attention:**

  Indicates the intensity of a user’s level of mental “focus” or “attention”, such as that which occurs during intense concentration and directed (but stable) mental activity. Distractions, wandering thoughts, lack of focus, or anxiety may lower the Attention meter levels.

- **Meditation:**

  Indicates the level of a user’s mental “calmness” or “relaxation”.

  Meditation is related to reduced activity by the active mental processes in the brain, and it has long been an observed effect that closing one’s eyes turns off the mental activities which process images from the eyes, so closing the eyes is often an effective method for increasing the Meditation meter level.

  Distractions, wandering thoughts, anxiety, agitation, and sensory stimuli may lower the Meditation meter levels.

At least that’s how it’s supposed to work. We’ve found that the degree of mental control over the signal varies from person to person. Ian Cleary, a peer of ours at ITP, used the Mind Flex in a recent project. He reports that about half of the people who tried the game were able to exercise control by consciously changing their mental state.

The most reasonable test of the device's legitimacy would be a comparison with a medical-grade EEG. While we have not been able to test this ourselves, NeuroSky has published the results of such a comparison.

Their findings suggest that the the NeuroSky chip delivers a comparable signal. Of course, NeuroSky has a significant stake in a positive outcome for this sort of test.

And there you have it. If you'd like to develop hardware or software around this data, I recommend reading the documentation that comes with the brain library for more information — or browse through the visualizer source to see how to work with the serial data.

If you make something interesting using these techniques, I'd love to hear about it.

March 2013 Update:

Almost three years on, I think I need to close the comments since I don't have the time (or hardware on hand) to keep up with support. Please post future issues on the GitHub page of the relevant project:

Arduino Brain Library
https://github.com/kitschpatrol/Arduino-Brain-Library

Processing Brain Grapher
https://github.com/kitschpatrol/Processing-Brain-Grapher

Most issues I'm seeing in the comments seem like the result of either soldering errors or compatibility-breaking changes to the Processing and Arduino APIs.

I'll try to stay ahead of these on GitHub and will be happy to accept pull requests to keep the code up to date and working. Thanks everyone for your feedback and good luck with your projects.

Visualizing Mind Activity With A Hacked Toy EEG

Hardware

The MindFlex toy includes a headset that reads EEG wave data. There's lots of information on hacking this toy to create a simple EEG machine using the limited data the headset provides by default. In this project I have extracted the full RAW EEG data using the same toy headset from the MindFlex game.

Friendly Warning And Disclaimer: Hack your toys at your own risk. I'm not responsible for any damage you do to yourself or your belongings. This information is provided here for information and education only. If you use the original hack and just use the data available in normal mode then the MindFlex still works as a game as originally intended. If you enable RAW EEG output then you break the toy!

Reading Brainwaves

An obvious place for us to start on this project might have been with the excellent OpenEEG project, but while being open, it still has substantial costs and complexity in getting up and running. For this project we looked for a simpler approach.

EEG technology has been incorporated into various toys and household items before. There are various commercial options available based on NeuroSky's ThinkGear chip.

The original Frontier Nerds blog post from 2010 provides a great starting point, but I've made a few changes along the way, including successfully extracting the RAW EEG values from the MindFlex toy.

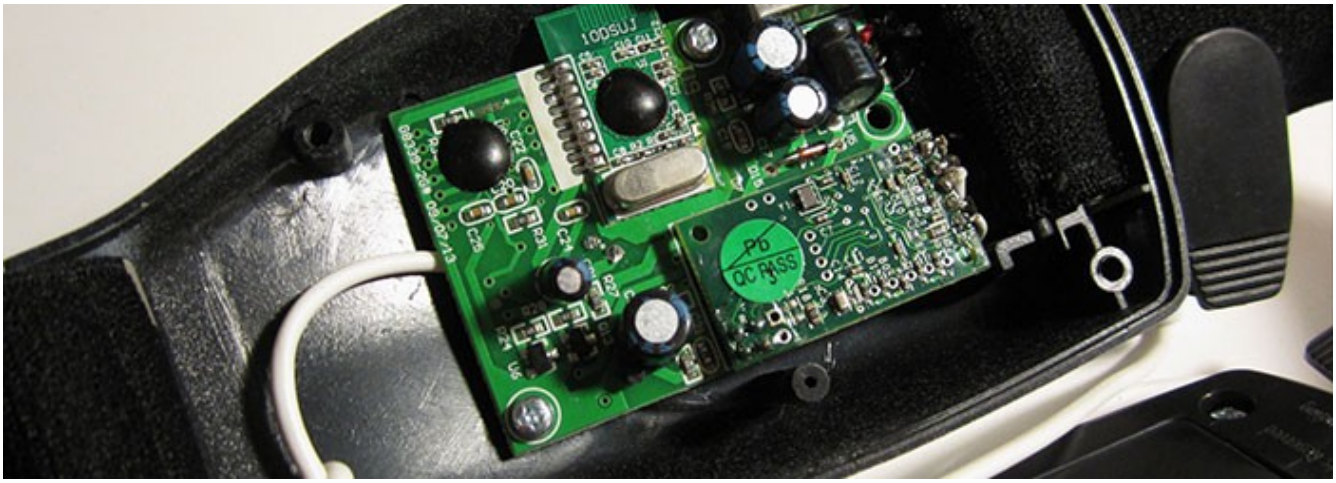
Budget EEG With The TGAM1

The TGAM1 is the controller board responsible for processing the EEG signals in all of the products based on the NeuroSky ThinkGear chip.

NeuroSky no longer sell the boards in small quantities (I've had confirmation of this from their business development team that this is due to support resources) and instead recommend that you buy the BrainWave starter kit.

This is probably a good idea if you want easy access to EEG data, but they are still over 100 GBP to buy in the UK, and the toys are available for only 40.

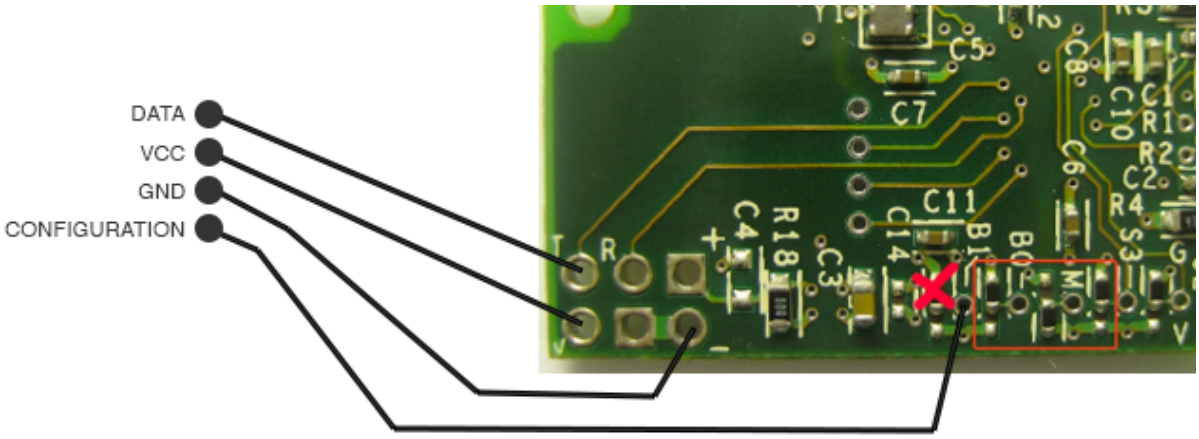
When you open the left compartment on the MindFlex you will see this:



The small board located on the top at the bottom right is the TGAM1 board.

The basic hack needs *just two solder points*. You solder on to the **T** pin to get the data out, and another connection for a common ground between your Arduinos circuit and the TGAM1.

I have also soldered two more connections on, which I will explain later.



Simple EEG Reading

The basic data that comes out of the MindFlex headset by default actually gives us quite a lot of information for only £40. In the default configuration you have soldered on only two connections, one for a common ground and one for the serial data stream.

I connect this to a SoftwareSerial pin on the Arduino and read in the packets. Here's what data you get:

- **Signal strength (0 - 200)**, where 0 means good signal, and 200 indicates a problem with the connection.
- **Attention and Meditation eSense values**, which are a proprietary measure that is calculated by the TGAM1 using NeuroSky's own algorithm. The toys based on this technology use the attention measure.
- **EEG power band values,** which are a measure of the various power bands that has been pre-calculated by the chip, this updates once per second.

The Arduino Brain Library already includes functionality for working with these values.

TGAM1 Region Change

If you bought the EU version and you're in EU then skip this bit.

The MindFlex, MindSet, Necomimi, etc all based on the TGAM1 that includes a notch filter on the EEG wave to remove electrical noise from the data.

If you bought a MindFlex from a different region then you will need to adapt it to work for you. For example, if you've bought a headset from the US it will have a notch filter at 60Hz. In Europe our "mains hum" is at 50Hz. You can switch the notch filter to work at the correct frequency with a bit of soldering.

Getting Full RAW EEG Data

Now on to getting the full raw EEG data from the MindFlex headset. In addition to the two connections you soldered on before there are two extra connections required.

This allows you to configure the TGAM1 board that is in the MindFlex to switch between two modes.

The 'configure' connection is actually called B1 on the TGAM1 board. In the MindFlex headset B1 is connected to GND via a 10K resistor. I have marked this resistor with an X on the image above.

- You need to remove this 10K resistor from the circuit board, and connect B2 to VCC via a 10K resistor to enable full raw EEG data output.
- The full EEG data is output at the higher baud rate of 57600. You also get the same packets as in normal mode.

You can switch back to normal mode, with just the basic data output at baud rate of 9600 by connecting B1 back to GND with the 10K resistor instead of the VCC connection.

Signal Strength

The normal data and raw data modes both give you the summary packet once every second.

In this is a measure of *signal strength*. It should be zero to indicate that the board is getting good measurements. Anything >0 indicates poor signal, and 200 indicates a problem.

When I had the Arduino hooked up to the computer serial port I struggled to get the signal strength to 0. When running off batteries (I have an Arduino Pro mini connected to the headset's battery pack) I have had no issues with signal strength.

The issues were resolved by adding an extra GND connection from the Arduino and connecting to the body of the person being tested. However, I don't recommend this approach; running the headset *completely off batteries* is recommended.

This requires no wired connection to the computer, hence the RF link that has been implemented.

RF Data Link

I am using a cheap low-power RF transceiver to send the EEG packets to the computer. This makes the headset wireless and solves the signal strength problem that required the extra ground connection.

Receiving The Data

The Processing Brain Grapher is a good starting point to check everything is working, *but it only works when the headset is in normal mode and sending the data via the serial port*.

I am working on a Processing library that will work with the updated headset in full EEG mode; this giving access to the eSense meters, the EEG power band values, and the full EEG raw data.

I'll be expanding this section as I make more progress with the libraries and visualisations.

September 2013 Update: I've posted an update, where I provide details of how to extract the full RAW EEG data from the MindFlex headset (covered below).

Full RAW EEG Data From The MindFlex Headset Hack

Hardware

There have been many hacks posted online over the past few years that show how to build a simple EEG machine using the Mattel MindFlex headset.

It's simple to interface this with the Arduino to read the basic single signal EEG data the toy produces. The toy actually only uses one of the measures (attention) but it outputs various useful bits of information.

All of this is detailed in the original Frontier Nerds blog post and on our Brain Wave Visualisation project page.

The original hacks are all based around the preprocessed data that the TGAM1 board in the MindFlex toy produces. I have successfully modified one of these headsets to get **full raw EEG data** from it.

I'm in the process of writing a library and updating the project page with full details, but in the meantime here's how it works.

Full Raw EEG Values

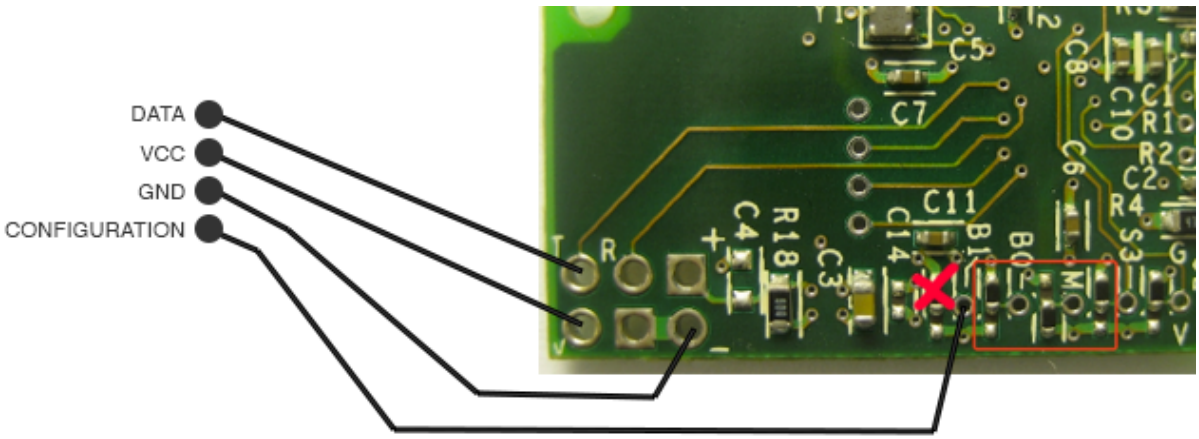
The TGAM1 board that is inside the MindFlex can run in three modes. One of these is 9600 bits per second normal mode which is how the MindFlex is configured by default.

This outputs a packet of data approximately every second that includes signal strength, attention level, meditation level, and pre-calculated values for 8 EEG power bands.

Friendly Warning and Disclaimer: Hack your toys at your own risk. I'm not responsible for any damage you do to yourself or your belongings. This information is provided here for information and education only.

If you use the original hack and just use the data available *in normal mode* then the MindFlex still works as a game as originally intended. If you enable RAW EEG output then you **break the toy**!

To enable RAW EEG you need to remove a 10K resistor from the TGAM1 board, and solder an extra connection to the board, as shown in this image:



The Hack

- Remove the 10K resistor that appears where the red X is on the photo.

You should double check with a multi-meter that you're removing the correct part. It is connected to the B1 port. This B1 port forms part of the configuration of the TGAM1 board.

In the MindFlex this 10k resistor connects the B1 port to GND which configures it in normal mode at 9600 bps.

- Solder a connection on to the B1 port, then you can add a new 10K resistor that connects the B1 port to V_{CC}. This enables raw output mode at 57600 bps.

You connect to the Arduino in the same way, *but you can't use the Arduino Brain Library with this configuration*. I'm working on an update for this.

The TGAM1 chip still sends the summary data packets that you get in normal mode once per second, but now you will also get a packet containing the raw EEG reading **512 times per second**, approximately once every 2ms.

Hacking MindWave Mobile

Attribution: Based on the SparkFun tutorial "Hacking MindWave Mobile" originally written by Sophi Kravitz via CC BY SA, and masterfully remixed by Mark Maffei.

This tutorial will explore the inner-workings of hacking the NeuroSky MindWave Mobile.

Project Scope

In this project, the analog brainwaves enter a processing ASIC chip and have digital values that are communicated over Bluetooth. I accessed only the digital data.

It is certainly possible to look at the analog brain waves before they enter the processing ASIC, but it would be much more difficult, requiring a schematic and specialized oscilloscope probes.

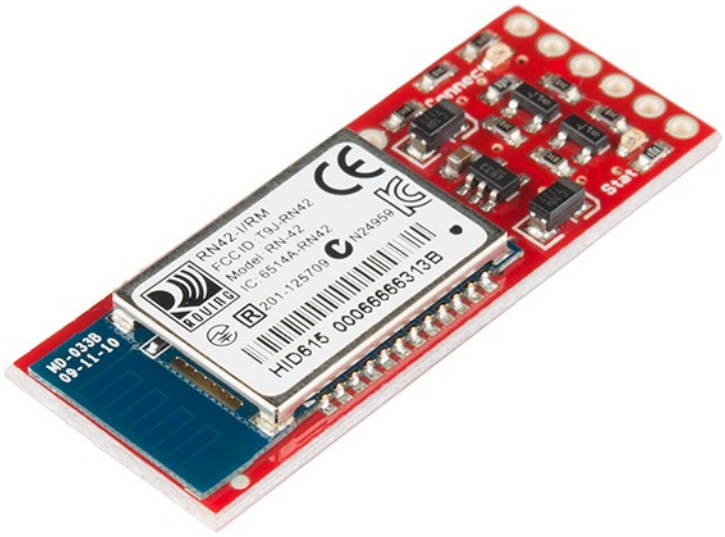
The Mindwave Mobile was chosen because it uses Bluetooth, making it easier to interface it with a microcontroller or other hardware.

Hardware

To interface with the MindWave, the RN-42 Bluetooth module was chosen. For this project, I created a custom PCB.

However, you could also use a BlueSMiRF or a Bluetooth Mate. The Bluetooth module will be connected to an Arduino Uno to read in the data being transmitted wirelessly. Once you've decided on which hardware you'll be using, connect everything.

The SparkFun Bluetooth Basics can come in quite handy if you're new to this.



The BlueSMiRF Silver module from SparkFun

Software

To read data from and configure the RN42 Bluetooth module:

- X-CTU, CoolTerm, or another serial terminal program of your choice.
- RS232 Port Logger

If unfamiliar with serial terminal emulators, check out SparkFun's terminal tutorial.

Firmware

- Here is the Arduino code.
- Here is the Processing code (interprets the MindWave data coming into the Arduino and gives you a visual representation of that data).

Sophi sporting her MindWave

Configuring The Bluetooth Module

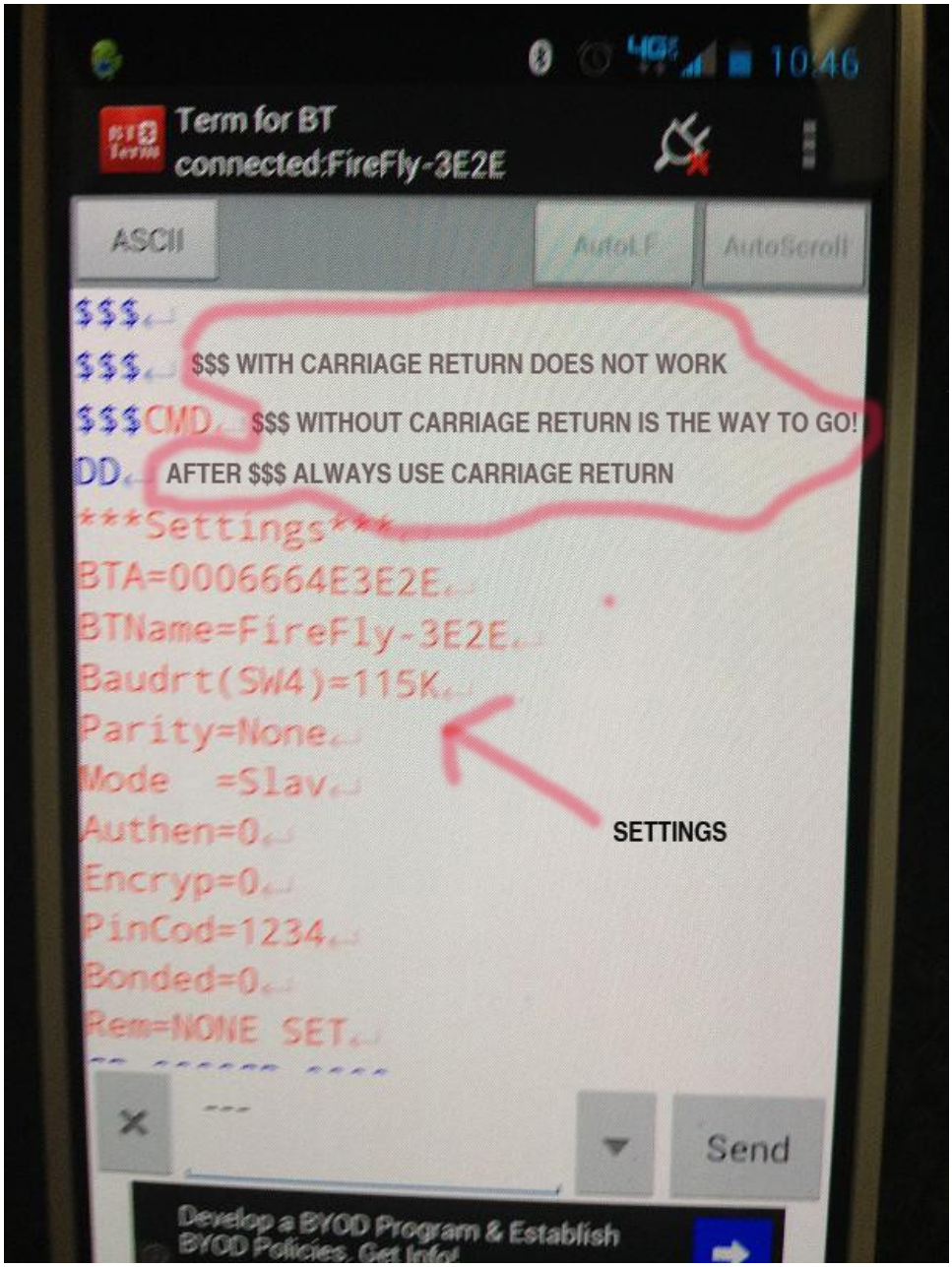
The RN42 module can be configured to behave in a variety of different ways. If you have never configured a RN Bluetooth module, scope out SparkFun's BlueSMiRF Hookup Guide.

You can use any serial terminal you wish to configure the Bluetooth module. For this project, I used my Android phone with an app called S2 Terminal for Bluetooth to configure the RN42. The configuration process went as so:

Note: If you are using the S2 Terminal program, you will need to type in ASCII and put a return after each command. Also, you only have 60 seconds after power up to enter the first command.

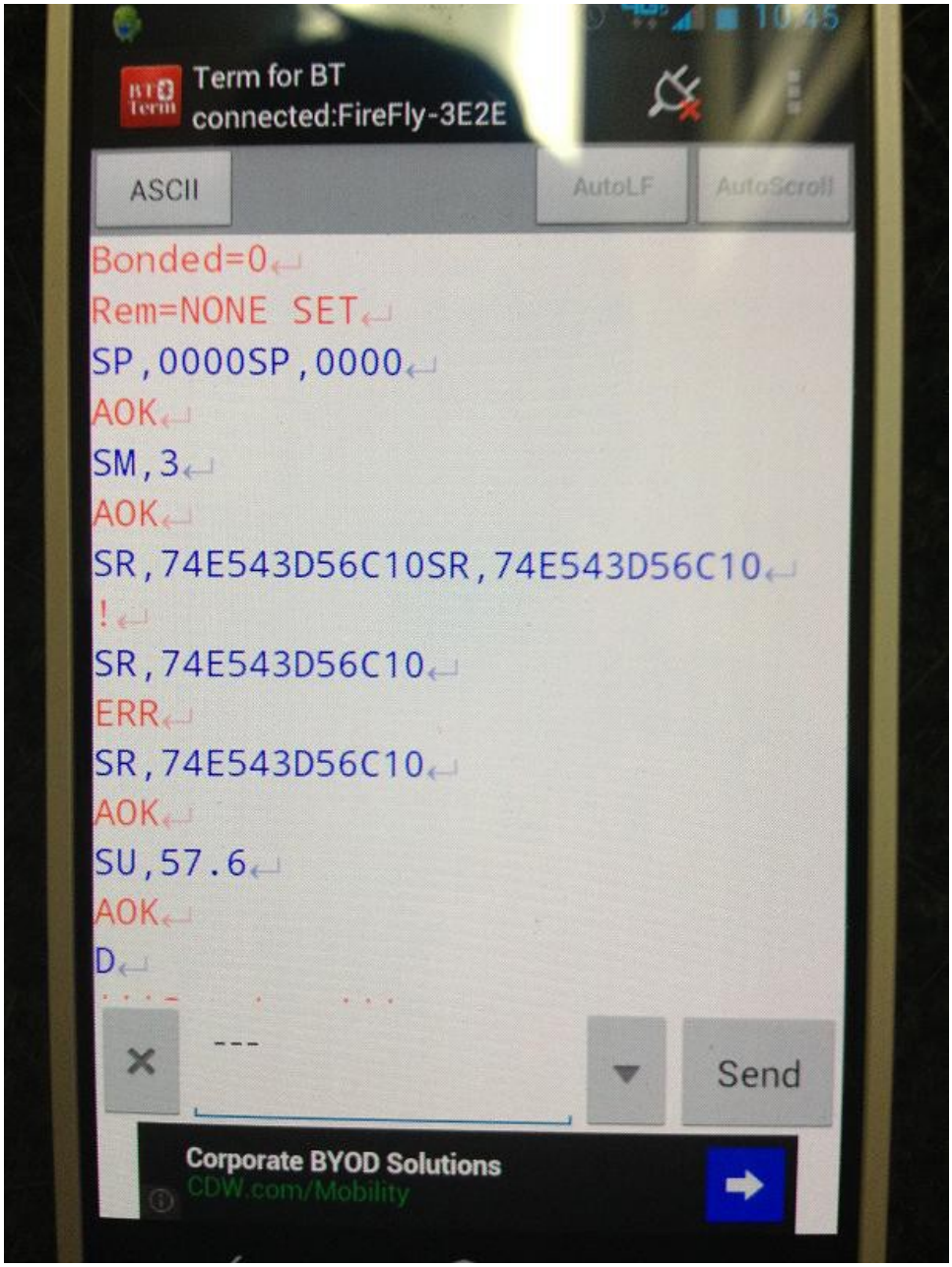
First:

1. To put the RN42 into COMMAND mode, type $$$ (NO carriage return). If successful, you will see "CMD".
2. Type: `D` + carriage return. You will see the current configuration of the RN42.



Next:

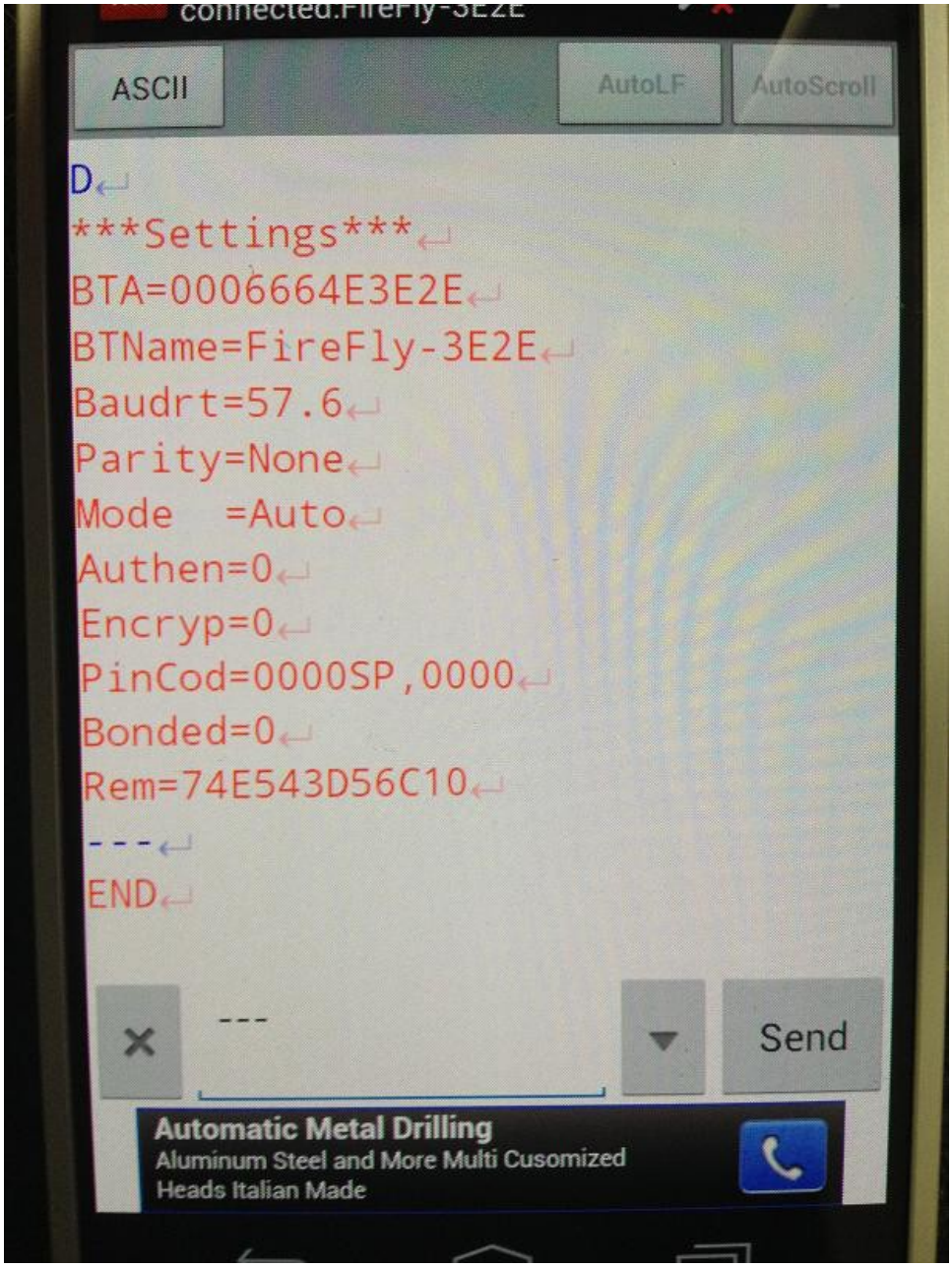
3. Type: `SP,0000` + carriage return. This will change the pincode from '1234' to '0000'. You will see "AOK" if this is done properly.
4. Type: `SM, 3` + carriage return. This will configure the RN42 to Auto-Connect Mode. Once the module is powered up, it will immediately look to connect (pair).You will see "AOK" if this is done properly.
5. Type: `SR,MAC ADDRESS` + carriage return. Insert the 12 digit address you copied from the MindWave Mobile. Look for AOK. If you don't see AOK, you'll have to retype the MAC address command.
6. Now type: `SU,57.6` + carriage return. This will change the baud rate from 115200 to 57600.



Then:

---

7. Type `D` + carriage return. Double check that the stored address is the Mac address you entered in step 5 and that it's configured to Auto, not Slave.
8. Type: `---` (three minus signs) + carriage return You should then see END.



Finally:

Turn the MindWave on. The blue light flashes for a few moments, then it will pair with your hardware. You'll know that the MindWave Mobile is paired because the blue light on the MindWave Mobile is solid.

If it is solid red, it is out of battery (AAA, lasts for about 8 hours).

Deciphering the MindWave Data

Using an FTDI basic, I was able to view the data coming in from the Arduino through my USB port. I used X-CTU to read the data.

The Arduino code linked above is mostly a combination of the Mindwave Mobile example code and Shane Clements' massaging. The code is set up to print all of the digital data available out of the MindWave Mobile. The data runs through the software serial port and is read easily on the X-CTU screen.

You can see unitless values for Delta, Theta, Alpha, Low Beta, High Beta and Gamma waves.

The MindWave's ASIC also presents a calculated value for both Attention and Meditation.

A number between 20 and 100 for Attention is normal, and a value above 40 means you are concentrating. I was able to get repeatable values with the Attention value by counting backwards from 100 to1.

We had a Bring A Hack dinner in Boulder, CO, and a few other people were able to make the Attention value rise by counting backwards from 100 to 1 also. What is interesting is that the 5 types of brainwaves measured and reported originate from different parts of the brain. They also have extremely low amplitudes and frequencies.

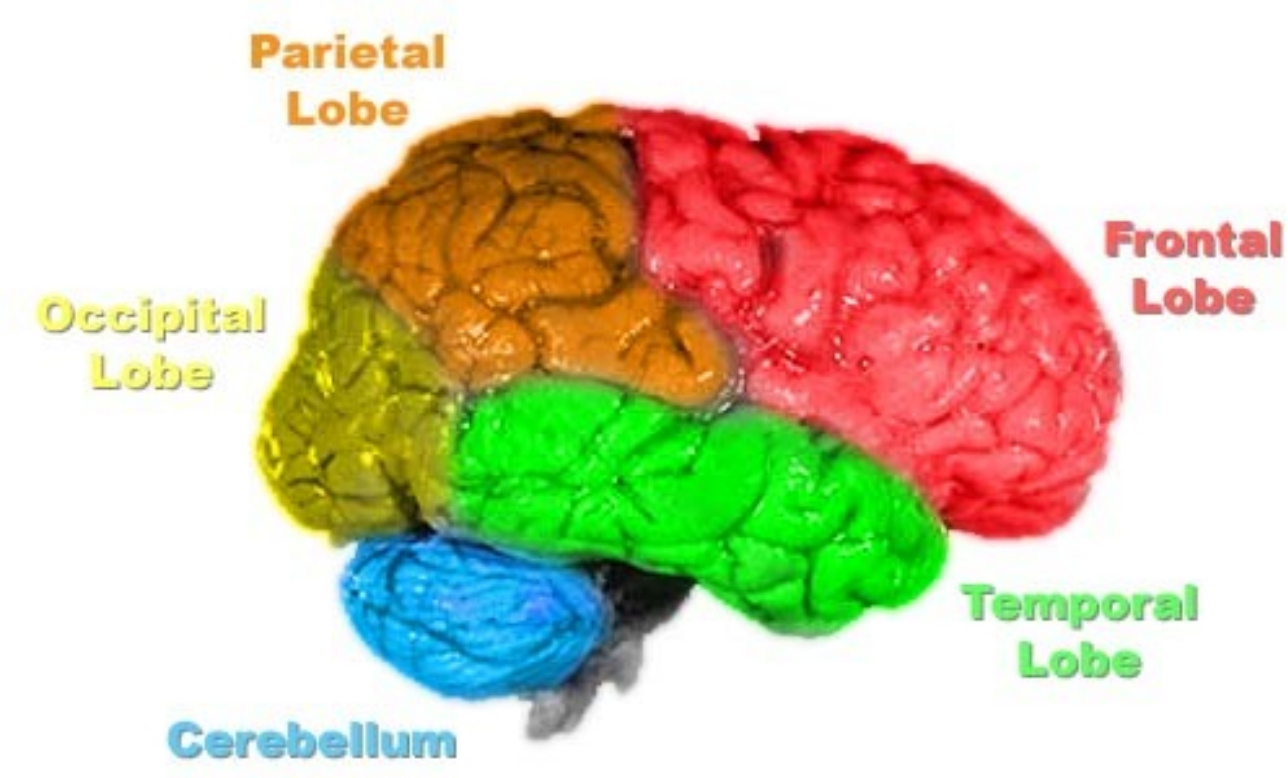
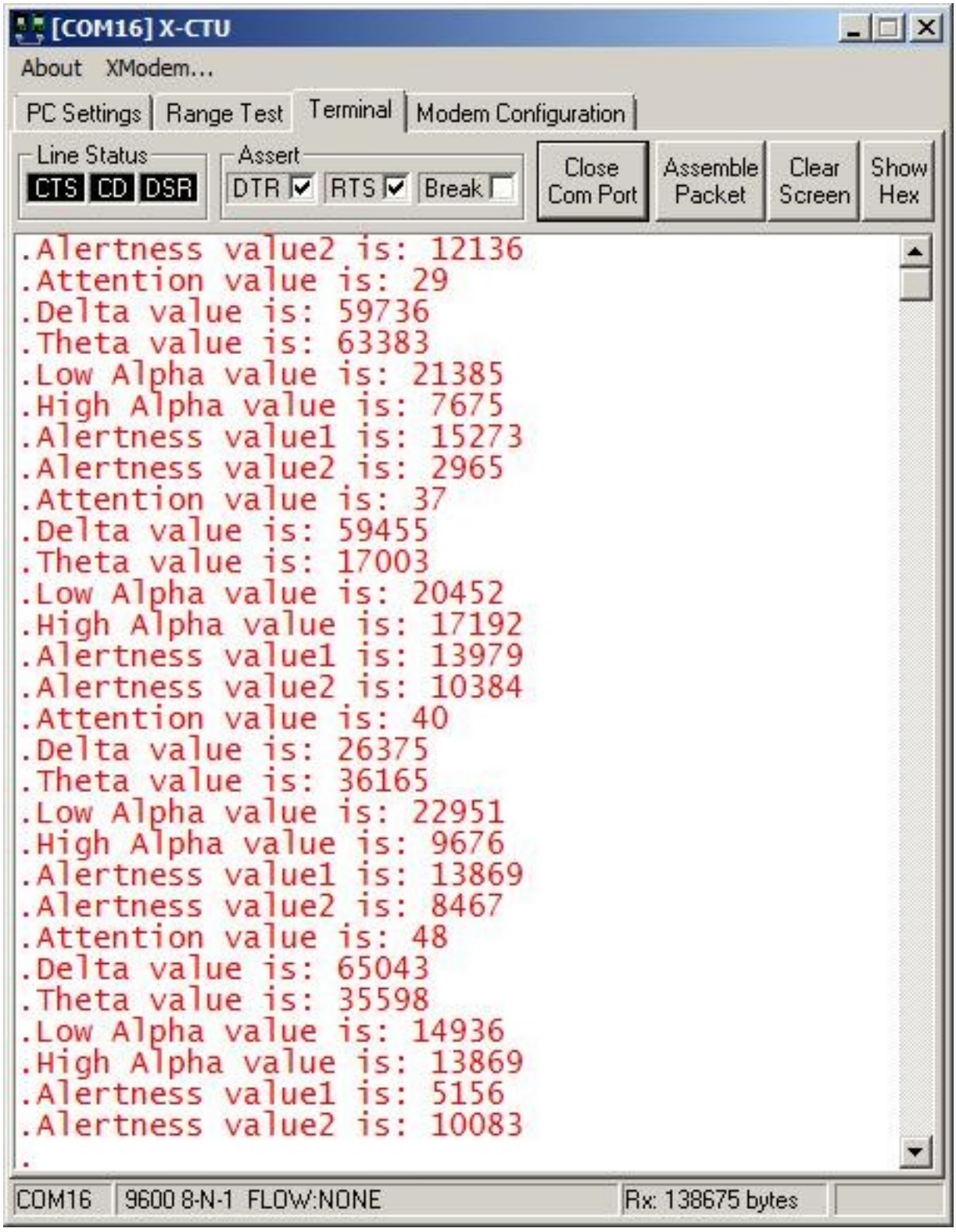
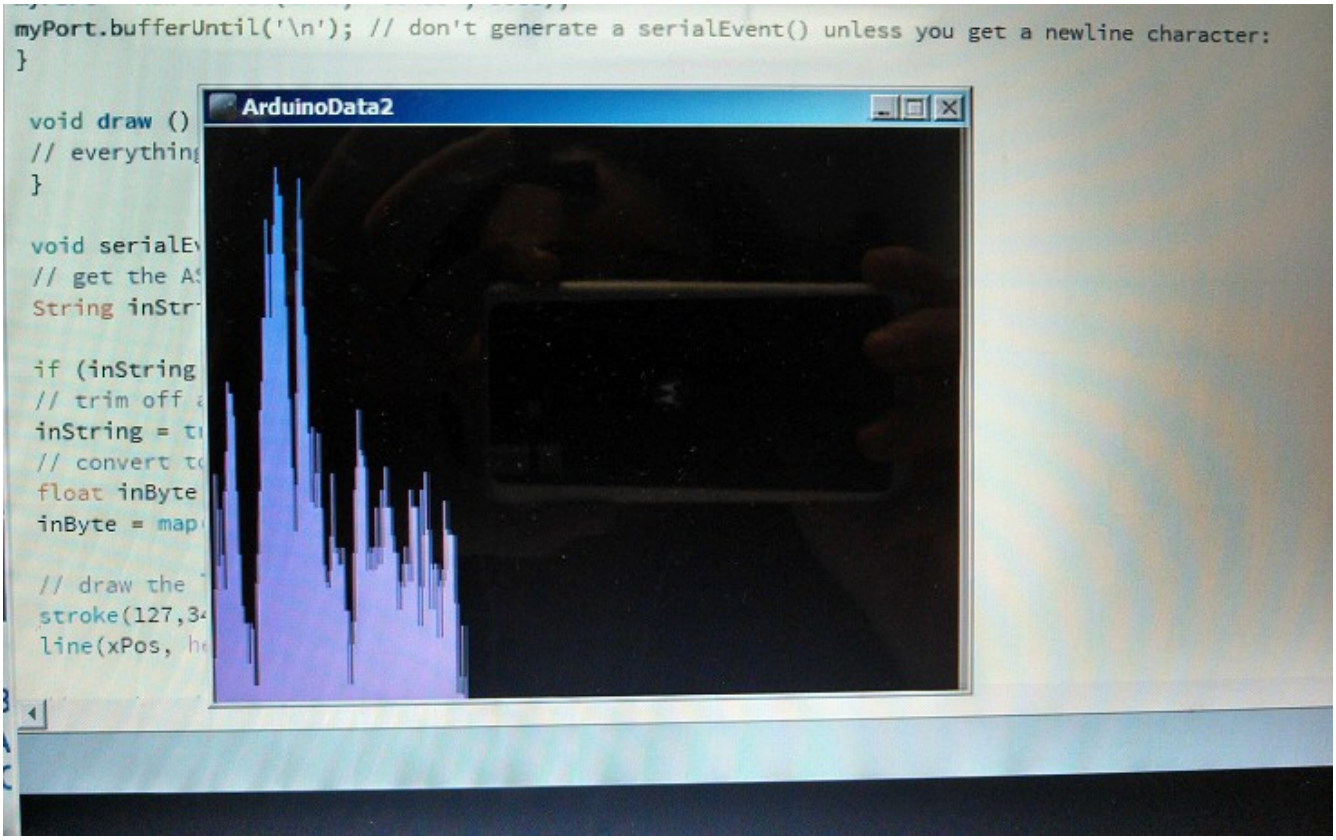


Image via Wikimedia.org

If you used the Arduino code linked above, you should be able to see these values printed out in X-CTU or the Serial terminal program of your choice:



The Processing code linked above can be used and modified to graph different values from the Arduino:



Conclusion

The best results using the MindWave Mobile came when I looked at the processed Attention value. It was easy to see that when I (or anyone else) counted backwards from 100 to 1, the Attention values went up.

The deep brainwave values that I really wanted to see, such as Gamma, Beta, etc., were all over the place, and it was difficult to see any consistency in those values at all.

In fact, the brainwave values varied the same when the unit was on the head as when off the head!

The Android app that comes with the MindWave Mobile didn't work at all, I tried it on a few phones without success. NeuroSky support thought that it was possible that my Android operating system was not compatible.

Overall, I don't think using the NeuroSky MindWave product yields any data that is very usable. I look forward to trying NeuroSky's next revision, and I am really looking forward to trying OpenBCI!

Pt. 5: Connecting Arduino to Processing

Attribution: Originally published by SparkFun, this highly streamlined adaptation of the following Processing tutorial was reconstructed by Mark Maffei via CC BY SA.

Introduction

As a serious Arduino brainhacker, having a basic understanding of how Arduino interacts with Processing is of great benefit. Here's the core essentials of what you need to know to get started.

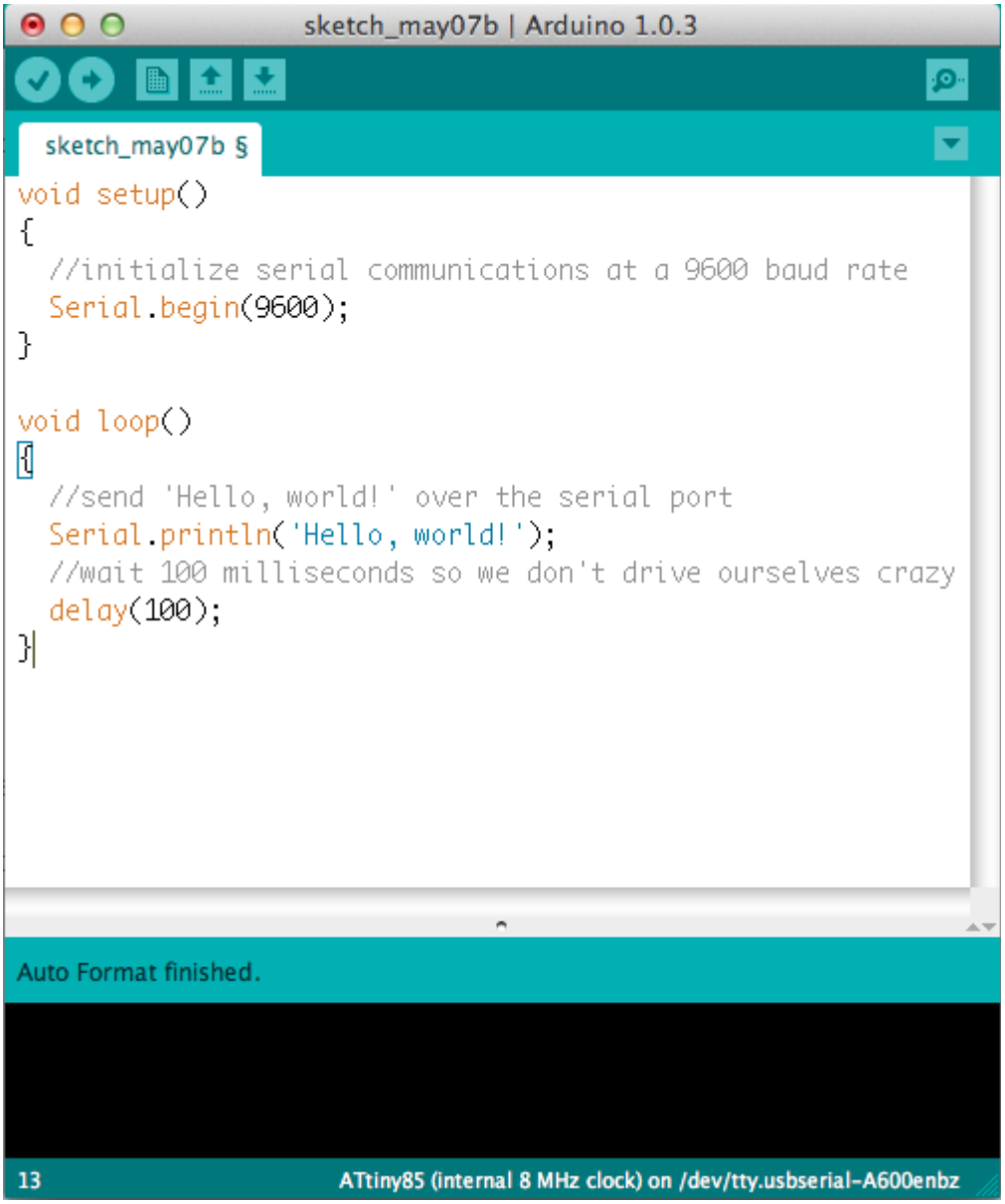
In this tutorial you will learn:

- How to send data from Arduino to Processing over the serial port
- How to receive data from Arduino in Processing
- How to send data from Processing to Arduino
- How to receive data from Processing in Arduino
- How to write a serial 'handshake' between Arduino and Processing to control data flow
- How to make a 'Pong' game that uses analog sensors to control the paddles
- While some basic familiarity with Processing will be useful, it's not strictly necessary.

Starting With The Arduino Side Of Things

Let's start with the basics of how to set up your Arduino sketch to send information over serial. Open up the Arduino software, and copy/paste the code below in.

After doing so, you should see something like this:



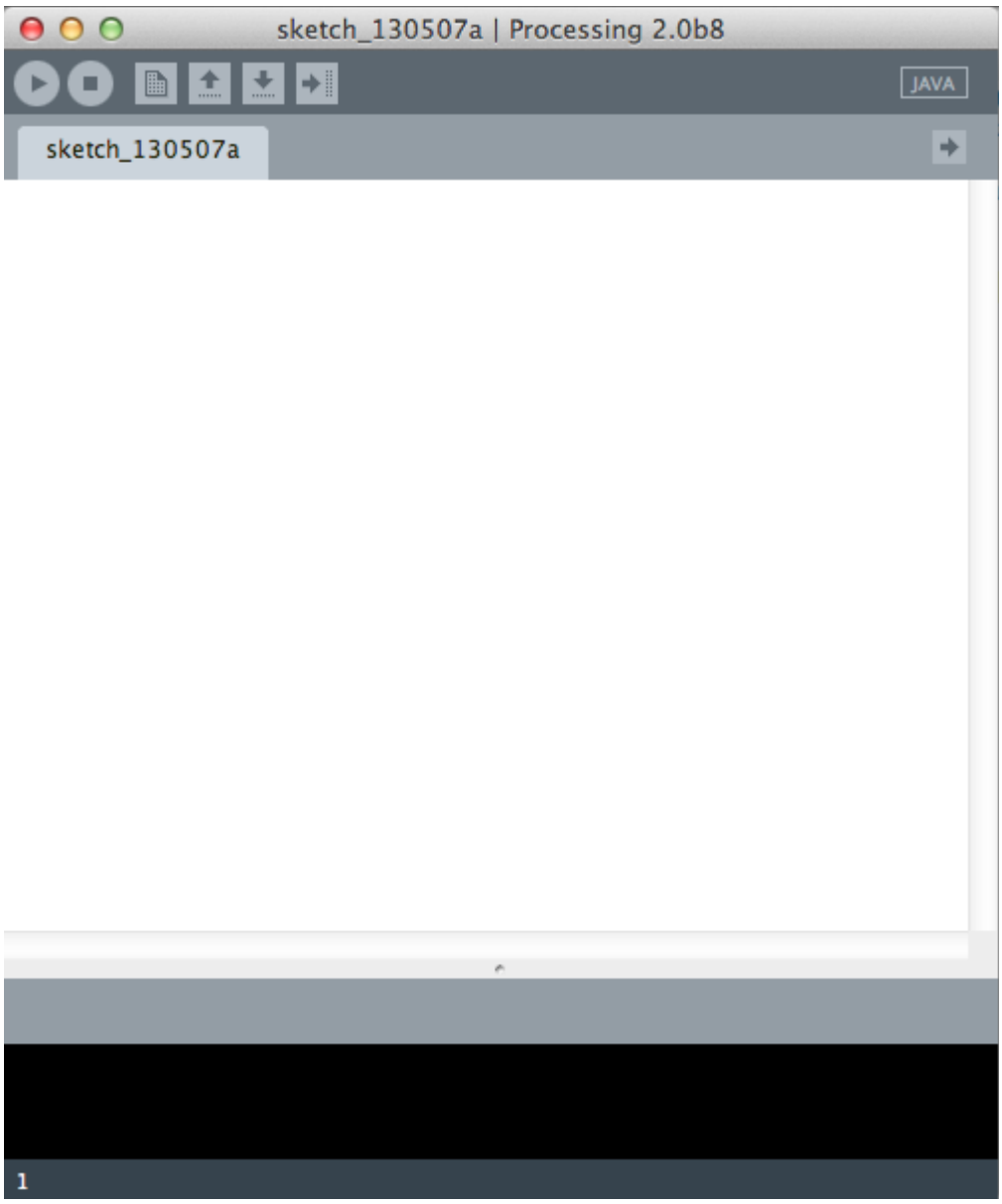
[Get Code Here]

All that's left to do is to plug in your Arduino board, select your board type (under **Tools -> Board Type**) and your Serial port (under **Tools -> Serial Port**) and hit the 'upload' button to load your code onto the Arduino.

Now we're ready to see if we can detect the 'Hello, world!' string we're sending from Processing.

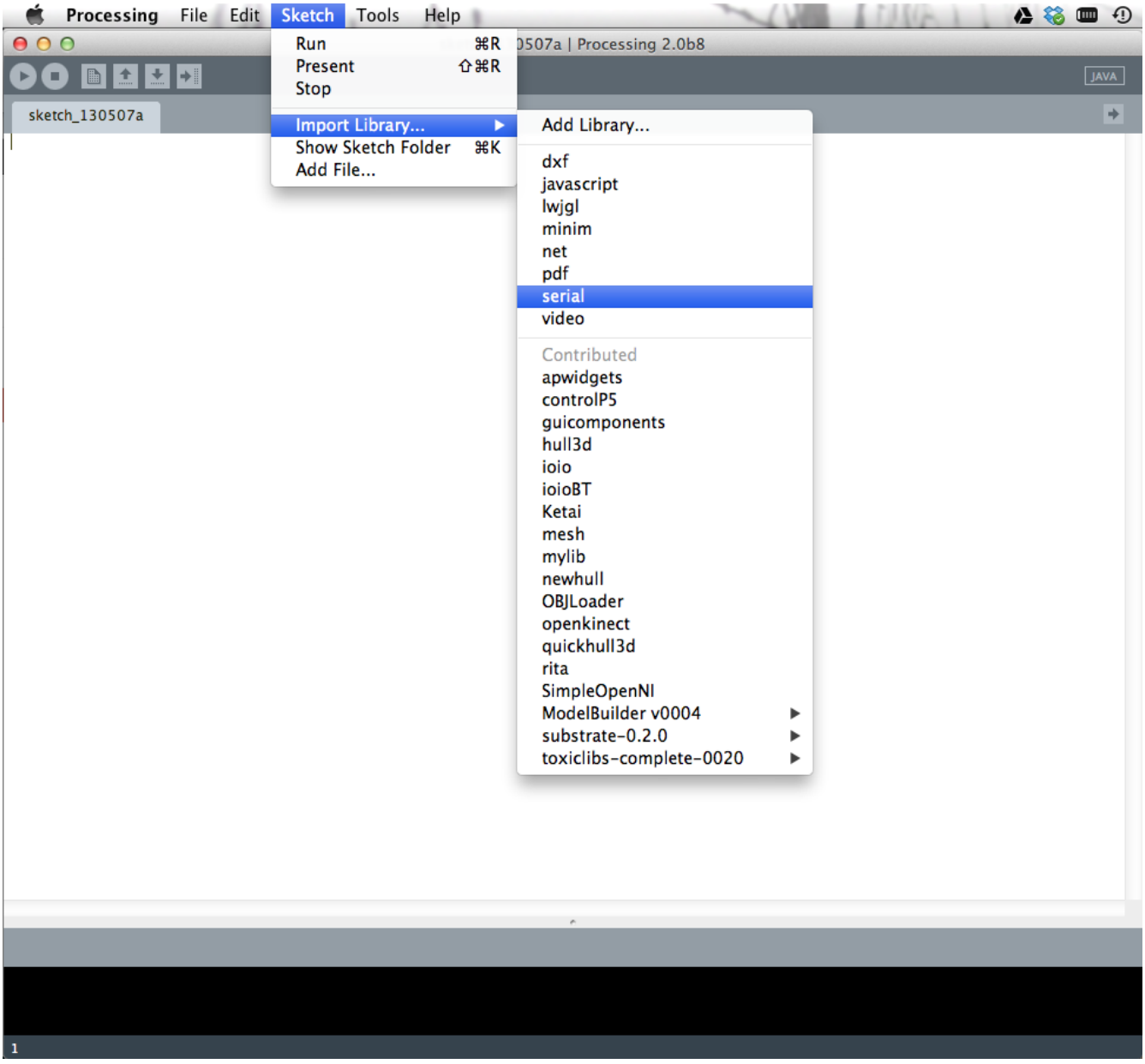
Communicating From Arduino To Processing

If you don't have a version of Processing, make sure you go to Processing.org and download the latest version for your OS. Once Processing is installed, open it up. You should see something like this:



If this looks a lot like the Arduino IDE, it's because the Arduino IDE was actually based in part off of Processing. Ah yes… the beauty of open source!

Ok, so once you have an open sketch, your first step is to import the Serial library. Go to **Sketch->Import Library->Serial**, as shown below:



You should now see a line like import processing.serial.*; at the top of your sketch.

Underneath your import processing.serial.*; at the top of your sketch, copy/paste the code example below (leaving a space between your import statement and the sample code).

A Brief Overview Of What's About To Happen

First, you'll declare some global variables. All this means is that these variables can used anywhere in your sketch. Add these two lines beneath the import statement in your sketch:

```
Serial myPort;  // Create object from Serial class
String val;     // Data received from the serial port
```

Now in order to listen to any serial communication, you'll have to get a Serial object (we call it myPort but you can it whatever you like), which lets you listen in on a serial port on your computer for any incoming data.

You'll also need a variable to receive the actual data coming in. In this case, since you're sending a String (i.e. the sequence of characters 'Hello, World!') from Arduino, you want to receive a String in Processing. Just like Arduino has setup() and loop(), Processing has setup() and draw() (instead of loop).

For your setup() method in Processing to work, you'll find the serial port your Arduino is connected to and set up your Serial object to listen to that port:

```
void setup()
{
  // I know that the first port in the serial list on my mac
  // is Serial.list()[0].
  // On Windows machines, this generally opens COM1.
  // Open whatever port is the one you're using.
  String portName = Serial.list()[0]; //change the 0 to a 1 or 2 etc. to match your port
  myPort = new Serial(this, portName, 9600);
}
```

Remember how you set Serial.begin(9600) in Arduino? Well, like the old saying "What you do to one side, you must also do to the other" definitely applies.

So you need to set 9600 as that last argument in your Serial object in Processing as well, allowing Arduino and Processing to communicate at the same rate.

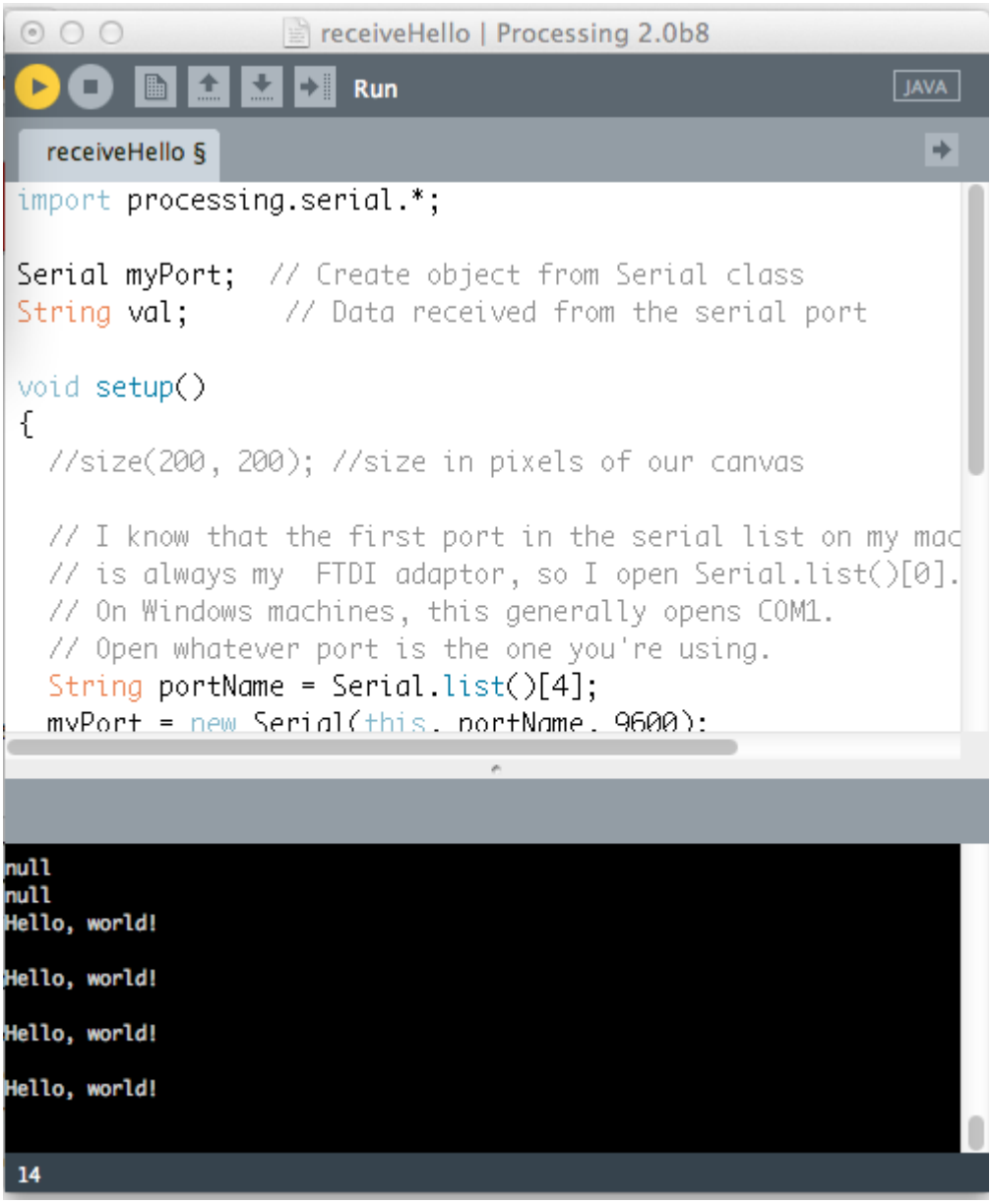
In your draw() loop, you're going to listen in on your Serial port and if you we get something, it'll go in your valvariable and print it to the console (that black area at the bottom of your Processing sketch)…

```
void draw()
{
  if ( myPort.available() > 0)
  {  // If data is available,
  val = myPort.readStringUntil('\n');         // read it and store it in val
  }
println(val); //print it out in the console
}
```

So now that you understand what's going on…

[Get Code Here]

If you hit the ‘run’ button (and your Arduino is plugged in with the code in the previous example loaded up), you should see a little window pop-up, and after a sec you should see `Hello, World!‘ appear in the Processing console. Over and over. Like this:



Assuming everything went according to plan… you’ve now conquered how to send data from Arduino to Processing. Your next step is figure out how go the opposite way - sending data from Processing to Arduino.

Communicating From Processing To Arduino

The Processing side of things starts out much like your last sketch: you import the Serial library and declare a global Serial object variable for your port up top. Likewise, in your setup() method you find your port and initialize Serial communication on that port with your Serial variable at 9600 baud.

You'll also use the size() command, to give yourself a little window to click in; which will trigger your sketch to send something over the Serial port to Arduino:

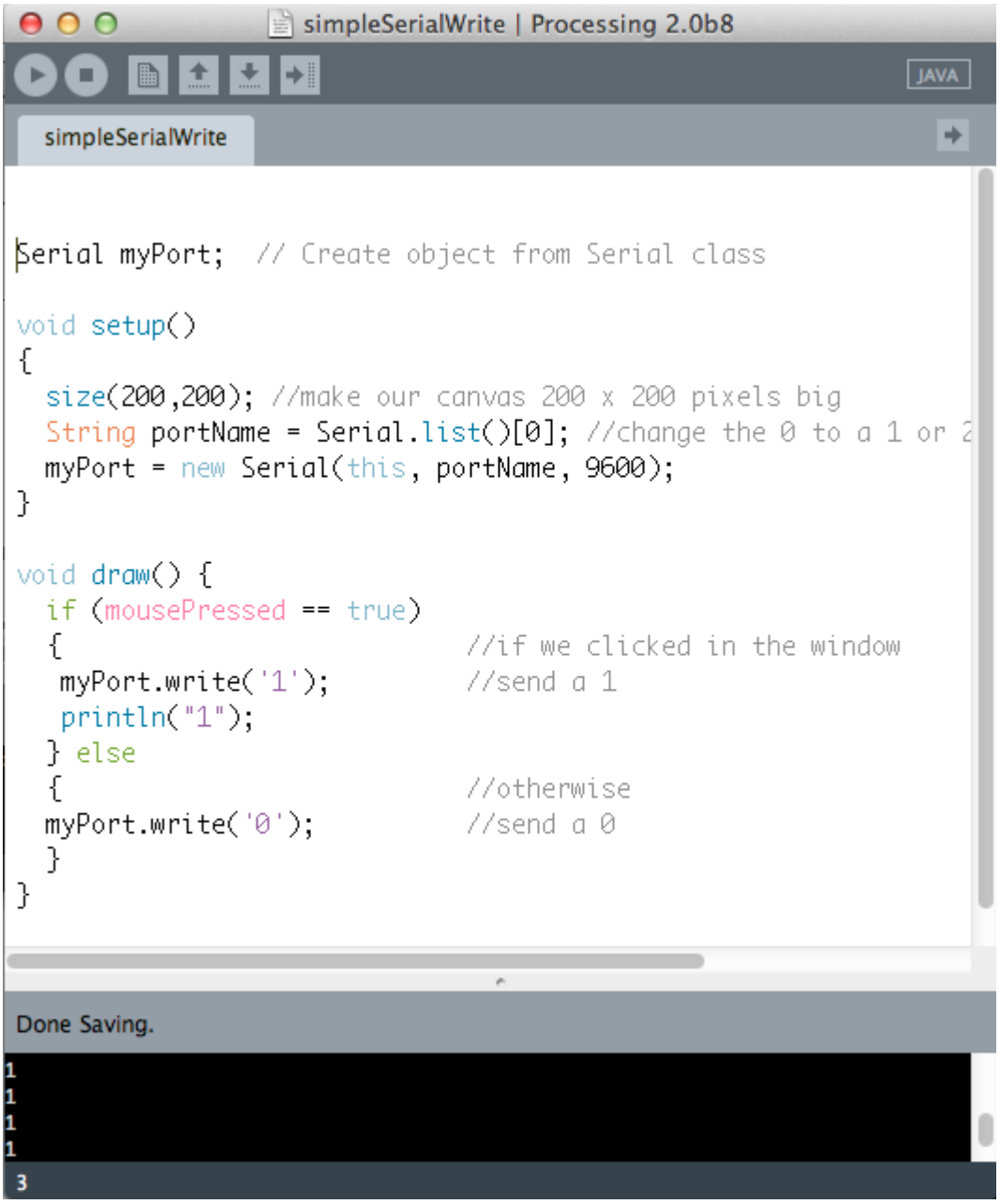
```
import processing.serial.*;

Serial myPort;  // Create object from Serial class

void setup()
{
  size(200,200); //make our canvas 200 x 200 pixels big
  String portName = Serial.list()[0]; //change the 0 to a 1 or 2 etc. to match your port
  myPort = new Serial(this, portName, 9600);
}
```

In your draw() loop, you send whatever you want over the serial port by using the write method from the Processing Serial library. For this sketch, you'll send a '1' whenever you click your mouse in the Processing window. You'll also print it out on the console, just to see that you're actually sending something. If you aren't clicking you'll send a '0' instead:

```
void draw() {
  if (mousePressed == true)
  {                       //if we clicked in the window
   myPort.write('1');         //send a 1
   println("1");
  } else
  {                       //otherwise
  myPort.write('0');          //send a 0
  }
}
```

[Get Code Here]

This is what your code should look like at this point:



If you run this code, you should see a bunch of 1's appear in the console area whenever you click your mouse in the window. But how do you look for these 1's from Arduino… and more importantly - what can you do with them?!

Glad you asked!

Coming Into Arduino

Your Arduino is going to look for those 1's coming in from Processing, and when it sees one, it'll turn on an LED on pin 13 (assuming that your Arduino has pin 13 designated as the on-board LED). This way, you won't need an external LED and resistor to see this work.

At the top of our Arduino sketch, we need two global variables - one for holding the data coming from Processing, and another to tell Arduino which pin our LED is hooked up to:

```
char val; // Data received from the serial port
int ledPin = 13; // Set the pin to digital I/O 13
```

Next, in your setup() method, you'll set the LED pin to an output (since you're powering an LED), and you'll once again start Serial communication at 9600 baud.

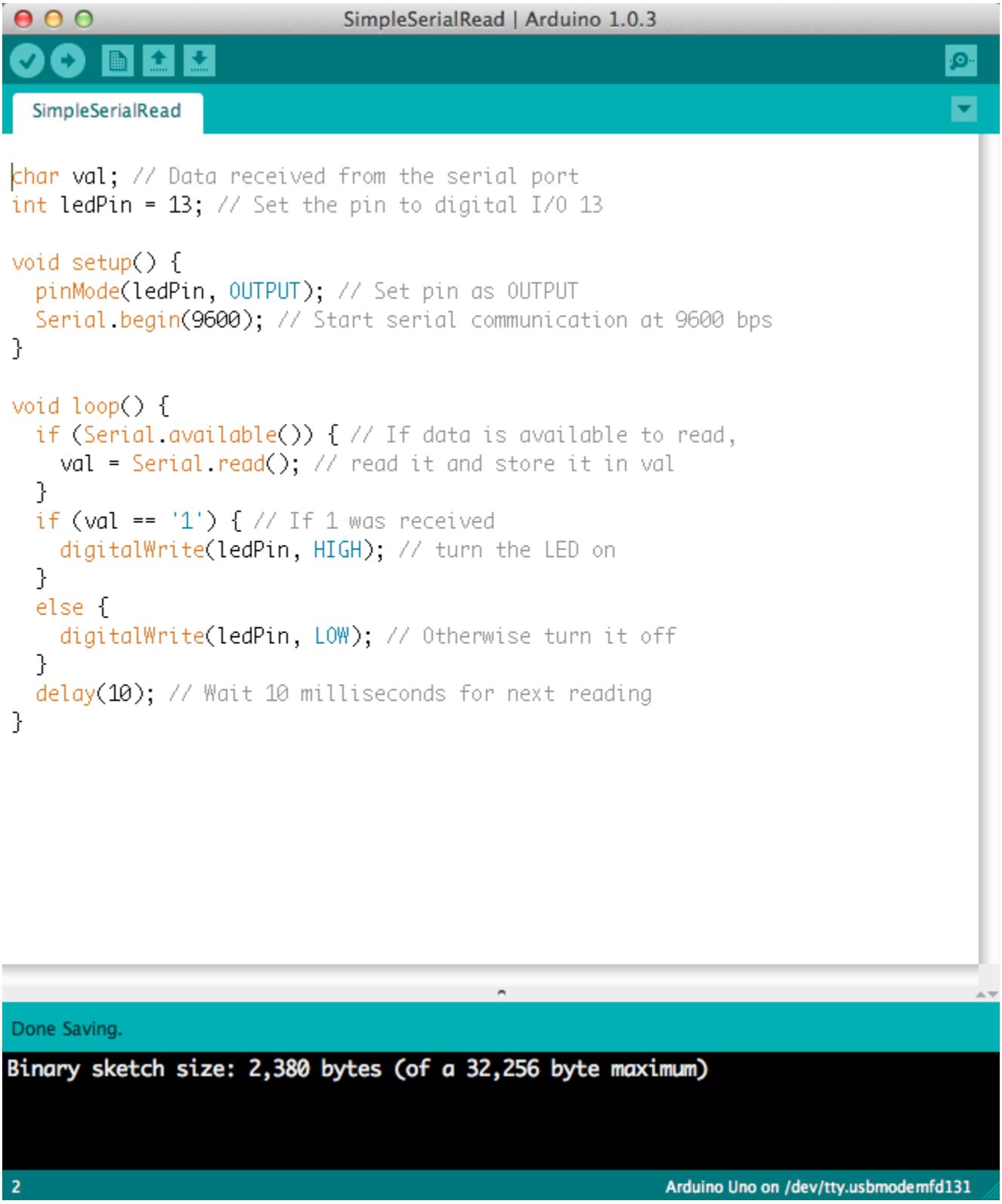
```
 void setup() {
   pinMode(ledPin, OUTPUT); // Set pin as OUTPUT
   Serial.begin(9600); // Start serial communication at 9600 bps
 }
```

Finally, in the loop() method, Arduino looks at the incoming serial data. If a '1', the LED gets set to HIGH (on). If a '0' - the LED gets set to off. At the end of the loop, there's a small delay to help Arduino keep up with the serial stream:

```
 void loop() {
   if (Serial.available())
   { // If data is available to read,
     val = Serial.read(); // read it and store it in val
   }
   if (val == '1')
   { // If 1 was received
     digitalWrite(ledPin, HIGH); // turn the LED on
   } else {
     digitalWrite(ledPin, LOW); // otherwise turn it off
   }
   delay(10); // Wait 10 milliseconds for next reading
}
```

[Get Code Here]

This is what your code should look like, all said and done:



If you load up this code onto your Arduino, and run the Processing sketch from the previous example, and everything goes smoothly… Voila! You should be able to turn on an LED attached to pin 13 of your Arduino, simply by clicking within the Processing canvas.

Shaking Hands

So far we've shown that Arduino and Processing can communicate via serial when one is talking and the other is listening. However, what about making a link that allows data to flow **both** ways; so that Arduino and Processing are *both sending and receiving data*?

It's called a serial 'handshake'; since both sides have to agree when to send/receive data.

From The Arduino Side

We'll now combine our two previous examples in such a way that Processing can both receive 'Hello, world!' from Arduino AND send a 1 back to Arduino to toggle an LED. Of course, this also means that Arduino has to be able to send 'Hello, world!' while listening for a 1 from Processing.

Let's start with the Arduino side of things. In order for this to run smoothly, both sides have to know what to listen for and what the other side is expecting to hear. We also want to minimize traffic over the serial port so we get more timely responses.

Just like in our Serial read example, we need a variable for our incoming data and a variable for the LED pin we want to light up:

```
char val; // Data received from the serial port
int ledPin = 13; // Set the pin to digital I/O 13
boolean ledState = LOW; //to toggle our LED
```

Since we're trying to be efficient, we're going to change our code so that we only listen for 1's, and each time we hear a '1' we toggle the LED on or off. To do this we added a boolean (true or false) variable for the HIGH or LOW state of our LED. This means we don't have to constantly send a 1 or 0 from Processing, which frees up our serial port quite a bit.

Our `setup()` method looks mostly the same, with the addition of an `establishContact()` function which we'll get to later - for now just type it in.

```
void setup()
{
  pinMode(ledPin, OUTPUT); // Set pin as OUTPUT
  //initialize serial communications at a 9600 baud rate
  Serial.begin(9600);
  establishContact();  // send a byte to establish contact until receiver responds
}
```

In our loop function, we've just combined and slimmed down the code from our two earlier sketches. Most importantly, we've changed our LED code to toggle based on our new boolean value.

The '!' means every time we see a one, we set the boolean to the opposite of what it was before (so LOW becomes HIGH or vice-versa). We also put our 'Hello, world!' in an *else statement*, so that we're only sending it when we haven't seen a '1' come in.

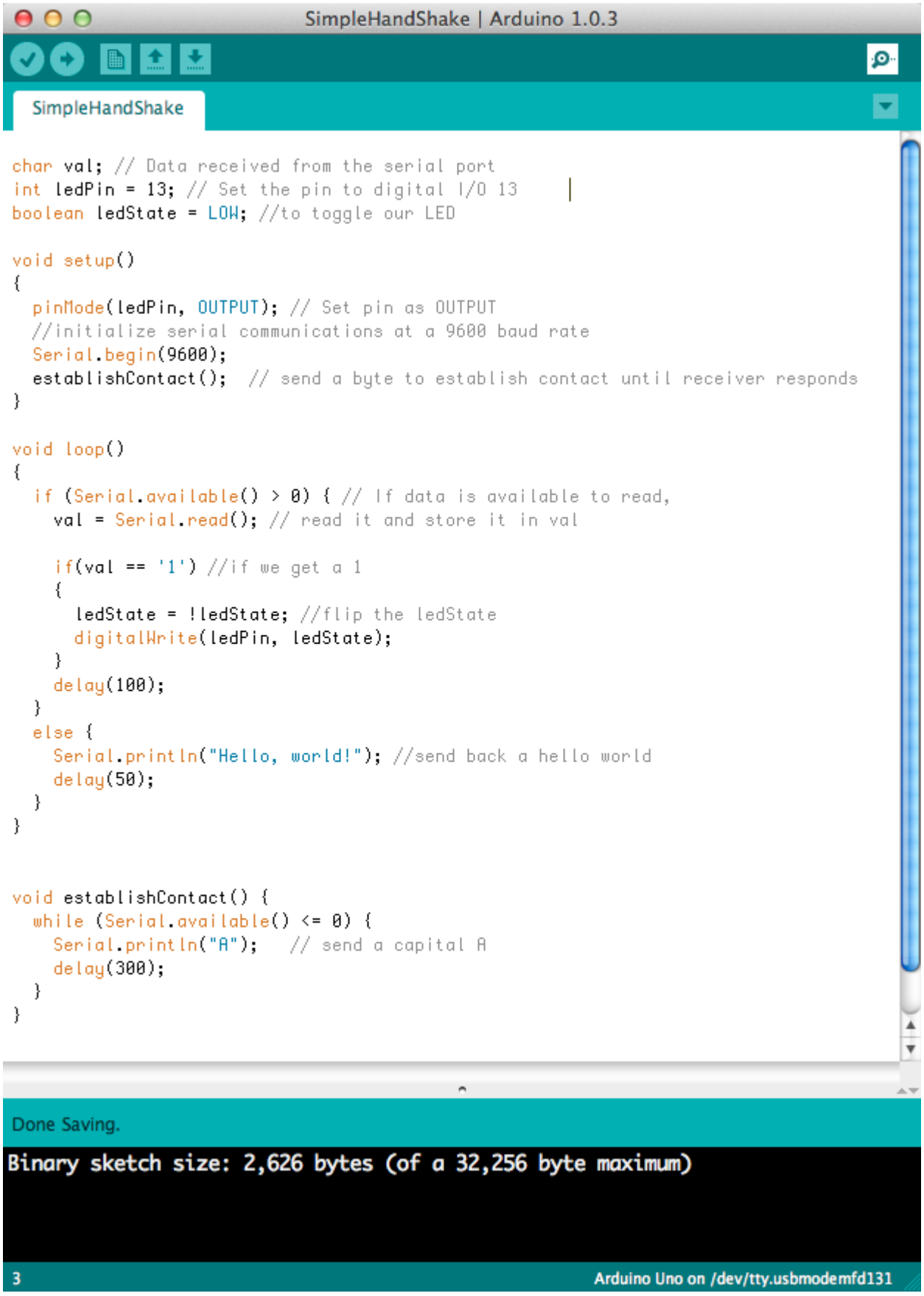
```
void loop()
{
  if (Serial.available() > 0) { // If data is available to read,
    val = Serial.read(); // read it and store it in val

    if(val == '1') //if we get a 1
    {
      ledState = !ledState; //flip the ledState
      digitalWrite(ledPin, ledState);
    }
    delay(100);
  }
    else {
    Serial.println("Hello, world!"); //send back a hello world
    delay(50);
    }
}
```

Now we get to that establishContact() function we put in our setup() method. This function just sends out a string (the same one we'll need to look for in Processing) to see if it hears anything back; indicating that Processing is ready to receive data. It's like saying 'Marco' over and over until you hear a 'Polo' back from somewhere.

```
void establishContact() {
  while (Serial.available() <= 0) {
  Serial.println("A");   // send a capital A
  delay(300);
  }
}
```

[Get Code Here]

Altogether, your Arduino code should look like this:


SimpleHandShake | Arduino 1.0.3
SimpleHandShake
char val; // Data received from the serial port
int ledPin = 13; // Set the pin to digital I/O 13
boolean ledState = LOW; //to toggle our LED
void setup()
{
pinMode(ledPin, OUTPUT); // Set pin as OUTPUT
//initialize serial communications at a 9600 baud rate
Serial.begin(9600);
establishContact(); // send a byte to establish contact until receiver responds
}
void loop()
{
if (Serial.available() > 0) { // If data is available to read,
val = Serial.read(); // read it and store it in val
if(val == '1') //if we get a 1
{
ledState = !ledState; //flip the ledState
digitalWrite(ledPin, ledState);
}
delay(100);
}
else {
Serial.println("Hello, world!"); //send back a hello world
delay(50);
}
}
void establishContact() {
while (Serial.available() <= 0) {
Serial.println("A"); // send a capital A
delay(300);
}
}
Done Saving.
Binary sketch size: 2,626 bytes (of a 32,256 byte maximum)
3
Arduino Uno on /dev/tty.usbmodemfd131

That's it for the Arduino side, now on to Processing!

From The Processing Side

For the Processing side of things, we've got to make a few changes. We're going to make use of the serialEvent() method, which gets called every time we see a specific character in the serial buffer; which acts as our delimiter - basically it tells Processing that we're done with a specific 'chunk' of data - in our case, one 'Hello, world!'.

The beginning of our sketch is the same except for a new firstContact boolean, which let's us know when we've made a connection to Arduino:

```
import processing.serial.*; //import the Serial library
 Serial myPort;  //the Serial port object
 String val;
// since we're doing serial handshaking,
// we need to check if we've heard from the microcontroller
boolean firstContact = false;
```

Our setup() function is the same as it was for our serial write program... *except* we added the myPort.bufferUntil('\n'); line. This let's us store the incoming data into a buffer, until we see a specific character we're looking for.

In this case, it's a carriage return (\n) because we sent a Serial.print *in* from Arduino. The 'ln' at the end means the String is terminated with a carriage return, so we know that'll be the last thing we see:

```
void setup() {
  size(200, 200); //make our canvas 200 x 200 pixels big
  //  initialize your serial port and set the baud rate to 9600
  myPort = new Serial(this, Serial.list()[4], 9600);
  myPort.bufferUntil('\n');
}
```

Because we're continuously sending data, our serialEvent() method now acts as our new draw() loop, so we can leave it empty:

```
void draw() {
  //we can leave the draw method empty,
  //because all our programming happens in the serialEvent (see below)
}
```

Now for the big one: serialEvent(). Each time we see a carriage return, this method gets called; hence we need to do a few things each time to keep things running smoothly:

- Read the incoming data and see if there's actually anything in it (i.e. it's not empty or 'null')
- Trim whitespace and other unimportant stuff
- If it's our first time hearing the right thing, change our firstContact boolean and let Arduino know we're ready for more data
- If it's *not* our first run, print the data to the console and send back any valid mouse clicks (as 1's) we got in our window
- Finally, tell Arduino we're ready for more data

That's a lot of steps, but luckily for us Processing has functions that make most of these tasks pretty easy. Let's take a look at how it all breaks down:

```
void serialEvent( Serial myPort) {
//put the incoming data into a String -
//the '\n' is our end delimiter indicating the end of a complete packet
val = myPort.readStringUntil('\n');
//make sure our data isn't empty before continuing
if (val != null) {
  //trim whitespace and formatting characters (like carriage return)
  val = trim(val);
  println(val);

  //look for our 'A' string to start the handshake
  //if it's there, clear the buffer, and send a request for data
  if (firstContact == false) {
    if (val.equals("A")) {
      myPort.clear();
      firstContact = true;
      myPort.write("A");
      println("contact");
    }
  }
  else { //if we've already established contact, keep getting and parsing data
    println(val);

    if (mousePressed == true)
    {                       //if we clicked in the window
      myPort.write('1');        //send a 1
      println("1");
    }

    // when you've parsed the data you have, ask for more:
```

```
    myPort.write("A");
    }
  }
}
```

Whew… that's a lot to chew on. But if you read carefully line by line (especially the comments), it'll start to make sense. If you've got your Arduino code finished and loaded onto your board, try running this sketch.

You should see 'Hello, world!' coming in on the console, and when you click in the Processing window, you should see the LED on pin 13 turn on and off. Success! You are now a serial handshake expert.

[Get Code Here]

Tips and Tricks

In developing your own projects with Arduino and Processing, there are a few 'gotchas' that are helpful to keep in mind in case you get stuck.

- Make sure your baud rates match.
- Make sure you're reading off the right port in Processing - there's a `Serial.list()` command that will show you all the available ports you can connect to.
- If you're using the `serialEvent()` method, make sure to include the `port.bufferUntil()` function in your `setup()` method.
- Also, make sure that whatever character you're buffering.
- If you want to send over a number of sensor values, it's a good idea to count how many bytes you're expecting so you know how to properly parse out the sensor data (the example shown below that comes with Arduino) gives a great example of this).

Going Further

Now that you know how to send data from Arduino to Processing and back again (even simultaneously!)… you're ready for some seriously cool projects.

By hooking together Arduino and Processing, you can do things like visualize sensor data in real-time, or make a glove with flex sensors in the fingers that makes penguins appear on the screen, or a command console from Processing that controls a giant array of LEDs.

Here are a few Processing links to increase your knowledge going forward:

- Derek Runberg's Processing Curriculum
- Arduino & Processing ATLAS Curriculum
- Processing the Danger Shield
- Arduino, Processing, & MaxMSP

Example From Above Tip: